2011-1

Science & Technology Facilities Council

Numerical Analysis Group Internal Report

C interfaces to HSL routines

J. D. Hogg

Version 1.0
5th December 2011

C interfaces to HSL routines

Jonathan Hogg

December 5, 2011

Abstract

The C programming language is widely used and can be interfaced to almost every serious numerical computational language in existence. By implementing a C interface to HSL routines the number of people able to use HSL software is significantly increased.

Interfacing C and Fortran 77 code in a portable fashion is a non-trivial problem that has a number of established solutions, such as the use of GNU autotools. Fortran 90 introduced modules and derived types that are not catered for by these approaches. Therefore extra attention is required to allow ready use of such software from C.

This guide is aimed at HSL developers and both describes how to use standards-compliant interoperability mechanisms for, and recommendations on the implementation of, consistent C interfaces to HSL routines.

Version	Date	Notes
1.0	5th December 2011	Original document

1 Outline

To implement a C interface to an HSL routine, the Fortran 2003 mechanism for interoperability with C should be used. This is described, for instance, in *"Modern Fortran Explained"*, by Metcalf, Reid and Cohen, and is supported by all modern Fortran compilers.

To avoid introducing a dependency on Fortran 2003 to the main Fortran code, the C interface is implemented through a separate wrapper. Hence, in addition to the main Fortran code (e.g. `hsl_ma97d.f90`), the C interface adds two additional files (for each precision):

- A C header file *<packagename><prec>*`.h` (e.g. `hsl_ma97d.h`) that specifies the C structures and function prototypes that form the C interface. It contains no executable code, and performs a function similar to the `.mod` file under Fortran (but must be written by hand rather than being automatically generated).
- A Fortran wrapper file *<packagename><prec>*`_ciface.f90` (e.g. `hsl_ma97d_ciface.f90`) that contains the Fortran definitions of the interoperable data types and wrapper routines described in the C header file. It will also often contain a module with helper routines that are used by more than one wrapper routine.

The C user must include the header in any file that makes a call to a HSL routine (similar to the Fortran `USE` statement). The C user must also link against the Fortran wrapper file and main Fortran code. If the Fortran compiler is not used to invoke the linker, the C user may need to explicitly include any Fortran compiler libraries in the link command. For C interoperability to work correctly, **matching C and Fortran compilers should be used**. If this is not done, the results are undefined.

Arguments and structures for the C interface need not match the Fortran one. This allows some functionality to be omitted from the C interface if desired. However, it is helpful to both the developer and user if C and Fortran interface naming and usage match as closely as possible.

For each routine in the main Fortran code, the Fortran wrapper defines a similar C-interoperable wrapper routine with the same name. This exactly matches a C function prototype in the C header file. The wrapper routine will provide any required translation between C and Fortran, and include a call to the Fortran version of the routine.

While built-in data types such as `integer` and `real` can be passed directly through the wrapper function as an argument of the main Fortran routine, this is not the case for defined types unless they have the `bind` attribute. To avoid needing to make any changes to the main Fortran code, the wrapper function defines a new type for each derived type in the main Fortran code and copies the components to and for, as needed. This will exactly match a C `struct` defined in the C header file. This means that there are three locations where a type with a given name are defined: the C header and Fortran wrapper files (that must match), and the main Fortran code (that doesn't have to).

Finally, it should be possible for the C interface to be used in such a fashion that it only adds a small overhead that is independent of problem size, even if this is not the default (e.g. it may require the use of Fortran indexing for arrays in C).

2 Additional features required in the C interface

2.1 Additions relating to array indexing

The default for the C interface must be to use 0-based indexing. As HSL packages generally do not alter original user data, this will often necessitate the time and memory overhead of a copy. One or more additional members of the (C) control type may be added allowing the user to specify if they wish to use 1-based (Fortran) indexing to avoid these overheads, for example:

C control type

```
struct ma97_control_d {
   int f_arrays; // If true use 1-based indexing
   int other_param; // Does something else
   ...
};
```

Main code Fortran control type

```
type ma97_control
   integer :: other_param = default
end type ma97_control
```

2.2 Additions for control type initialisation

C does not offer any mechanism for setting default values of structure members. Therefore an additional routine must be added to the interface for a control type that sets its default values. This should be named `<packagename>_default_control`. See Section 5.1 for further information.

3 The header file

The header file specifies the interface to the C compiler. Additional short comments should be included to allow its use as a reference by the user.

Generally it consists of the following sections:

Copyright statement

Permission has been obtained to distribute the header files (only) under a modified BSD licence. This allows users to distribute them with their code and dynamically load HSL routines from a shared library if they are available. This functionality is exploited by Ipopt.

The copyright statement at the start of the file states these permissions.

#include guard

The C preprocessor lines

```
#ifndef HSL_MA97D_H
#define HSL_MA97D_H
...
#endif
```

are referred to as a #include guard. They ensure that, if the header file is included more than once by the user, symbols are not multiply defined.

#define symbol remapping

C does not allow generic calls in the same way as Fortran. Each routine must therefore have a unique name. HSL data types and functions therefore have an underscore followed by a single letter appended to their name indicating their type (e.g. `_d` for double precision).

The typical user will only require to use a single precision variant within their code. In this case, and to simplify documentation, it is useful to omit the precision specifying suffix. This is achieved by creating an alias through the use of the C pre-processor's #define directive, and only affects the C header file and user's program. By placing these directives in an #ifndef block we ensure only the first header file encountered defines these aliases.

While it is possible to exploit the fact that the first encountered header file is the one to define these aliases, it is not recommended. Instead, best practice for the user is to always use the explicit suffix in mixed precision codes.

```
/* Order of header file inclusion determines meaning of ma97_control
   and ma97_default_control() */
#include "hsl_ma97d.h"
#include "hsl_ma97s.h"

struct ma97_control dcontrol; /* same as ma97_control_d */
struct ma97_control_s scontrol;

ma97_default_control(dcontrol); /* same as ma97_default_control_d */
ma97_default_control_s(scontrol);
```

Typedefs

To minimise differences between header files for different precisions in a package, C `typedef`s should be used to provide an alias to define the package types. These names end with an underscore to indicate that they are expected to be internal to the header file.

Struct definitions

The structure definitions must match, in both order and types of members, the corresponding type definition in the Fortran interface file. The name of the structure and the names of its members need not match; this removes the need to append precision suffixes to types in the Fortran wrapper file, even though it is required in the C header file.

Function definitions

The C function prototype must match in data type and name the corresponding procedure definition in the Fortran interface file. Corresponding argument names may differ, but for ease of maintenance should be made to match.

Example

Listing 1 shows a skeleton C header file.

Listing 1: Skeleton of a C header file

```
/*
 * COPYRIGHT (c) 2011 Science and Technology Facilities Council (STFC)
 * Original date 20 September 2011
 * All rights reserved
 *
 * Written by: Jonathan Hogg
 *
 * THIS FILE ONLY may be redistributed under the modified BSD licence below.
 * All other files distributed as part of the HSL_MA97 package
 * require a licence to be obtained from STFC and may NOT be redistributed
 * without permission. Please refer to your licence for HSL_MA97 for full terms
 * and conditions. STFC may be contacted via hsl(at)stfc.ac.uk.
 *
 * Modified BSD licence (this header file only):
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *  * Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *  * Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *  * Neither the name of STFC nor the names of its contributors may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL STFC BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
 * OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */

#ifndef HSL_MA97D_H
#define HSL_MA97D_H

#ifndef ma97_default_control
#define ma97_control ma97_control_d
...
#endif

typedef double ma97pkgtype_d_;
typedef double ma97realtype_d_;

struct ma97_control_d {
    int f_arrays;              /* Use C or Fortran numbering */
    int action;                /* Continue on singularity if !=0 (true),
                                  otherwise abort */
    ma97realtype_d_ u;         /* Pivoting parameter */
    ...
};

/* Set default values of control */
void ma97_default_control_d(struct ma97_control_d *control);
void ma97_enquire_indef(void **keep, ma97pkgtype_d_ d[]);
...
#endif
```

4 Fortran interface file

An example of a skeleton Fortran wrapper file is given in Listing 2.

Listing 2: Skeleton of a Fortran wrapper file

```
module hsl_ma97_double_ciface
   use iso_c_binding
   use hsl_ma97_double, only:                    &
      f_ma97_akeep          => ma97_akeep,       &
      ...
      f_ma97_control        => ma97_control
   implicit none

   integer, parameter :: C_DOUBLE

   type, bind(C) :: ma97_control
      integer(C_INT) :: f_arrays ! true(!=0) or false(==0)
      real(wp) :: u
   end type ma97_control

   ...

contains

subroutine copy_control_in(ccontrol, fcontrol, f_arrays)
   type(ma97_control), intent(in) :: ccontrol
   type(f_ma97_control), intent(out) :: fcontrol
   logical, intent(out) :: f_arrays

   f_arrays                  = (ccontrol%f_arrays.ne.0)
   fcontrol%u                = ccontrol%u
end subroutine copy_control_in

end module hsl_ma97_double_ciface

subroutine ma97_default_control_d(ccontrol) bind(C)
   use hsl_ma97_double_ciface
   implicit none

   type(ma97_control), intent(out) :: ccontrol

   type(f_ma97_control) :: fcontrol

   ccontrol%f_arrays = 0 ! false
   ccontrol%u = fcontrol%u
   ...
end subroutine ma97_default_control_d

...
```

4.1 Code organisation

To minimise the amount of repeated code used to remap names in the `use` statements and to specify derived types a module should be used to allow reuse of this code in the wrapper. It should have the name of the Fortran module it is wrapping with `_ciface` appended (e.g. `hsl_ma97_double_ciface`). This module may also contain subroutines that copy data between the types used in the main Fortran and the interoperable types used in the wrapper, along with any other common code.

4.2 Symbol naming conventions

Note that all names of routines and derived types that are imported from the main Fortran code in the `USE` statement are renamed to include the prefix `f_`. This avoids confusion as to which routine or data type we are using.

Variable names start with a `c` or `f` to indicate whether they refer to the C or Fortran variable (which may in fact be pointers to the same bit of memory). Where the C and Fortran references for a variable are the same (for example, those that will be `INTEGER, INTENT(IN)` dummy arguments of the Fortran routines) no prefix is used.

5 Handling HSL derived types and unsupported situations

In general, developers should refer to existing C interfaces or language reference materials to see how basic types are handled. This section covers HSL standard ways of handling our derived types and some common situations that are not supported under currently available standards.

5.1 A `control` type

An HSL `control` type is used to pass parameters that affect how an algorithm behaves. It has `intent(in)` with no allocatable components. For example, the main Fortran definition might be as follows:

```
type ma97_control
   logical :: action = .true.
   integer :: nemin = 8
   integer, dimension(2) :: lpage = 2**12
   double precision :: u = 0.01
end type ma97_control
```

A new interoperable data type is created that has a Fortran definition in *<packagename>*`_ciface.f90`:

```
type, bind(C) :: ma97_control_d
   use iso_c_binding
   integer(C_INT) :: action
   integer(C_INT) :: nemin
   integer(C_INT), dimension(2) :: lpage
   real(C_DOUBLE) :: u
end type ma97_control_d
```

and a C definition in *<packagename>*`.h`:

```
struct ma97_control {
   int action; // 0 is true, 1 is false
   int nemin;
   int lpage[2];
   double u;
};
```

Since (pre-C99) there is no logical data type in C, an integer is used instead, following the C convention that `0` is false and non-zero is true. Further, C has the concept of reserved words that may clash with component names. If this is the case, an underscore is appended to the name in the C version only. For example, `control.static_` matches `control%static`.

A simple Fortran routine can translate between the interoperable and main data types:

```
subroutine copy_control_in(ccontrol, fcontrol)
   type(ma97_control), intent(in) :: ccontrol
   type(f_ma97_control), intent(out) :: fcontrol

   fcontrol%action = (ccontrol%action.eq.0)
   fcontrol%neim = ccontrol%nemin
   fcontrol%lpage(:) = ccontrol%lpage(:)
   fcontrol%u = ccontrol%u
end subroutine copy_control_in
```

For each routine a wrapper may then be written:

```
subroutine ma97_do_something_d(ccontrol) bind(C)
   use hsl_ma97_double_ciface
   implicit none

   type(ma97_control), intent(in) :: ccontrol
```

```
    type(f_ma97_control) :: fcontrol

    call copy_control_in(ccontrol, fcontrol)
    call f_ma97_do_something(fcontrol)
end subroutine ma97_do_something_d
```

with a C prototype of

```
void ma97_do_something_d(const struct ma97_control *control);
```

To set default values a new routine is implemented in the Fortran wrapper file:

```
subroutine ma97_default_control_d(ccontrol) bind(C)
    use hsl_ma97_double_ciface
    implicit none

    type(ma97_control), intent(out) :: ccontrol

    type(f_ma97_control) :: fcontrol

    if(fcontrol%action) then
       ccontrol%action = 0 ! true
    else
       ccontrol%action = 1 ! false
    endif
    ccontrol%nemin = fcontrol%nemin
    ccontrol%lpage(:) = fcontrol%lpage(:)
    ccontrol%u = fcontrol%u
end subroutine ma97_default_control_d
```

This copies the defaults from the original Fortran type to future proof against any changes to them. We give it the following C prototype:

```
void ma97_control_initialize_d(struct ma97_control *control);
```

6 An `info` type

An HSL `info` type is used to return statistical information to the user. It has `intent(out)` and no allocatable components. They can be handled in a very similar fashion to `control` variables, except that there are no default values. For example, there might be the following definition of the main Fortran type:

```
type ma97_info
    integer(selected_int_kind(18)) :: nfactor
    integer :: nsup
    double precision :: detlog
end type ma97_info
```

The interoperable type will have the following C header file entry

```
struct ma97_info_d {
    long int nfactor;
    int nsup;
    double detlog;
};
void ma97_get_stats_d(struct ma97_info_d *info);
```

and the Fortran wrapper file will look similar to:

```
module hsl_ma97_double_ciface
   use hsl_ma97_double, only :: f_ma97_info => ma97_info, &
                                f_ma97_get_stats => ma97_get_stats
   use iso_c_binding
   implicit none

   type, bind(C) :: ma97_info
      use iso_c_binding
      integer(c_long) :: nfactor
      integer(c_int) :: nsup
      real(c_double) :: detlog
   end type ma97_info
contains
   subroutine copy_info_out(finfo, cinfo)
      type(f_ma97_info), intent(in) :: finfo
      type(ma97_info), intent(out) :: cinfo

      cinfo%nfactor = finfo%nfactor
      cinfo%nsup = finfo%nsup
      cinfo%detlog = finfo%detlog
   end subroutine copy_info_out
end module hsl_ma97_double_ciface

subroutine ma97_get_stats_d(cinfo) bind(C)
   use hsl_ma97_double_ciface
   implicit none

   type(ma97_info), intent(out) :: cinfo

   type(f_ma97_info) :: finfo

   call f_ma97_get_stats(finfo)
   call copy_info_out(finfo, cinfo)
end subroutine ma97_get_stats_d
```

7 A keep type

A `keep` variable is characterised as one used to preserve information between calls to routines of a package. Typically these store large amounts of data that the user should not alter. Further, the data layout is not made available to the user as it is complex and subject to future change, so is not documented to a standard suitable for publishing in a user guide. Often these data types have allocatable components that are not interoperable with C under currently available standards.

Handling these data types using a mixture of the techniques above for `control` and `info` variables may be feasible, but is undesirable because of potentially much larger copying overheads at run time in addition to increased demands on the programmer. Instead the `private` nature of the data type can be exploited as described below.

Consider the following typical example of a Fortran definition

```
type ma97_keep
   private
   integer :: nnodes
   type(ma97_node), dimension(:), allocatable :: nodes
end type ma97_keep
```

that is not interoperable, but does consider its components to be `private`. It can be treated as a black box

by the user, and as such an untyped pointer (`void *`) is used to store the location of the type in C. As C does not know the size of the structure or the format of its descriptor, it can only be allocated on the Fortran side. This means that the user must pass a pointer to this `void *` variable (i.e. a `void **`) so that it may be set by the Fortran code. As such a typical call may have the following prototype

```
void ma97_analyse_d(void **keep, const struct ma97_control_d *control,
      struct ma97_info_d *info);
```

while the Fortran side looks like this

```
subroutine ma97_analyse_d(ckeep, ccontrol, cinfo) bind(C)
   use hsl_ma97_double_ciface
   implicit none

   type(c_ptr), intent(out) :: ckeep
   type(ma97_control), intent(in) :: ccontrol
   type(ma97_info), intent(in) :: cinfo

   type(f_ma97_keep), pointer :: fkeep
   type(f_ma97_control) :: fcontrol
   type(f_ma97_info) :: finfo

   call copy_control_in(ccontrol, fcontrol)
   allocate(fkeep); ckeep = c_loc(fkeep)

   call f_ma97_analyse(fkeep, fcontrol, finfo)

   call copy_info_out(finfo, cinfo)
end subroutine ma97_analyse_d
```

On subsequent calls where `keep` has `intent(inout)` and is already allocated we may replace the allocate with a pointer association such as in the following:

```
subroutine ma97_factorise_d(ckeep, ccontrol, cinfo) bind(C)
   use hsl_ma97_double_ciface
   implicit none

   type(c_ptr), intent(inout) :: ckeep
   type(ma97_control), intent(in) :: ccontrol
   type(ma97_info), intent(in) :: cinfo

   type(f_ma97_keep), pointer :: fkeep
   type(f_ma97_control) :: fcontrol
   type(f_ma97_info) :: finfo

   call copy_control_in(ccontrol, fcontrol)
   call c_f_pointer(ckeep, fkeep)

   call f_ma97_factorise(fkeep, fcontrol, finfo)

   call copy_info_out(finfo, cinfo)
end subroutine ma97_factorise_d
```

Obviously, Fortran will need to free the memory as C has insufficient information to do so. This can often be handled transparently in the finalise routine:

```
subroutine ma97_finalise_d(ckeep) bind(C)
   use hsl_ma97_double_ciface
   implicit none
```

```
    type(c_ptr), intent(inout) :: ckeep

    type(f_ma97_keep), pointer :: fkeep

    call c_f_pointer(ckeep, fkeep)

    call f_ma97_finalise(fkeep)

    deallocate(fkeep); ckeep = c_null_ptr
end subroutine ma97_finalise_d
```

Note that we could use a `void *` rather than `void **` pointer after the first call. Instead, we maintain the `void **` data type to avoid confusion.

8 Nested types

Nested (interoperable) types are supported in the obvious fashion:

```
! Fortran
type, bind(C) :: inner
   integer(C_INT) :: data1
end type inner
type, bind(C) :: outer
   type(inner) :: inner1
   integer(C_INT) :: data2
end type
...
outer%inner1%data1 = 1

/* C */
struct inner {
   int data1;
};
struct outer {
   struct inner inner1;
   int data2;
};
outer.inner1.data1 = 1;
```

9 Optional arguments

Under Fortran 2003, interoperability of optional arguments is not supported. While a supplemental technical report to Fortran 2008 is under development to support this, this issue must be worked around at present.

For a Fortran routine with optional arguments there are three choices:

1. Do not support optional arguments in the C interface.
2. Use the convention on the C side that a `NULL` pointer means argument not present. The Fortran side will then need to test each argument before attempting to associate it. This may not be a good option if an argument has `intent(out)` and is not of interoperable type.

3. Offer different names in C for different combinations of optional arguments. Again, the Fortran side will need to handle this, but can do so using Fortran-style optional arguments and a non-C interoperable main routine that does most of the work.

e.g.
Main Fortran routine

```
subroutine opt_eg(arg1, arg2)
   integer, intent(inout) :: arg1
   integer, optional, intent(in) :: arg2

   arg1 = arg1**2
   if(present(arg2)) arg1 = arg1 + arg2**2
end subroutine opt_eg
```

Fortran interface with optional argument omitted.

```
! void opt_eg(int *arg1);
subroutine opt_eg(arg1) bind(C)
   use mymodule_ciface

   integer(C_INT) :: arg1

   call f_opt_eg(arg1)
end subroutine opt_eg
```

Fortran interface with optional argument via C convention

```
! void opt_eg(int *arg1, const int *arg2);
subroutine opt_eg(arg1, carg2) bind(C)
   use mymodule_ciface

   integer(C_INT) :: arg1
   type(C_PTR), value :: carg2

   integer(C_INT), pointer :: farg2

   nullify(farg2)
   if(C_ASSOCIATED(farg2)) call C_F_POINTER(carg2, farg2)

   if(associated(farg2)) then
      call f_opt_eg(arg1, arg2=farg2)
   else
      call f_opt_eg(arg1)
   endif
end subroutine opt_eg
```

Fortran interface with different names for with and without

```
! void opt_eg_without(int *arg1);
subroutine opt_eg_without(arg1) bind(C)
   use mymodule_ciface

   integer(C_INT) :: arg1

   call f_opt_eg(arg1)
end subroutine opt_eg_without

! void opt_eg_with(int *arg1, int arg2);
subroutine opt_eg_with(arg1, arg2) bind(C)
```

```
   use mymodule_ciface

   integer(C_INT) :: arg1
   integer(C_INT), value :: arg2

   call f_opt_eg(arg1, arg2=arg2)
end subroutine opt_eg_with
```

10 Strings

While the C and Fortran `char` and `character(len=1)` types are interoperable, the `char *` and `character(len=*)` types require more work. The following code may be of use:

```
module hsl_ma97_double_ciface
   use iso_c_binding
   implicit none

   interface
      integer(c_size_t) pure function strlen(cstr) bind(C)
         use iso_c_binding
         implicit none
         type(c_ptr), value, intent(in) :: cstr
      end function strlen
   end interface
contains
   function cstr_to_fchar(cstr) result(fchar)
      type(c_ptr) :: cstr
      character(kind=c_char,len=strlen(cstr)) :: fchar

      integer :: i
      character(c_char), dimension(:), pointer :: temp

      call c_f_pointer(cstr, temp, shape = (/ strlen(cstr) /) )

      do i = 1, size(temp)
         fchar(i:i) = temp(i)
      end do
   end function cstr_to_fchar
end module hsl_ma97_double_ciface

subroutine ma97_open_d(cname)
   use hsl_ma97_double_ciface
   implicit none

   type(c_ptr), intent(in) :: cname

   character( kind=c_char, len=strlen(cname) ) :: fname

   fname = cstr_to_fchar(cname)

   ...
end subroutine ma97_open_d
```

11 Documentation

A separate version of the user documentation should be produced for C programmers. This should involve relatively minor changes to the Fortran spec sheet. In particular the following changes should be made:

- Care should be taken as C is case sensitive. HSL style is that **all C names must be in lower case**.
- C `structs` have **members** not `components`. Members are denoted by `struct.member` not `type%component`.
- A new subsection entitled "**C interface to Fortran code**" should be added as the first subsection of the "How to use" section. Its wording should be similar to the following:

  > **2.1 C interface to Fortran code**
  > This package is written in Fortran and a wrapper is provided for C programmers. This wrapper implements a subset of the full functionality described in the Fortran user documentation. The wrapper will automatically convert between 0-based (C) and 1-based (Fortran) array indexing, so can be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing. The conversion is disabled by setting the control parameter `control.f_arrays=1` and supplying all data using 1-based indexing. With 0-based indexing, the matrix is treated as having rows and columns $0, 1, \ldots n-1$. **In this document, we assume 0-based indexing**.
  >
  > The wrapper uses the Fortran 2003 interoperability features. **Matching C and Fortran compilers must be used**, for example, gcc and gfortran, or icc and ifort. If the Fortran compiler is not used to link the user's program, additional Fortran compiler libraries may need to be linked explicitly.

- All `LOGICAL` variables should be replaced with C `int` variables. A value of 0 indicates false and any other value indicates true. The values of true or false should be explicit in the documentation, e.g. "If `check!=0` (i.e. evaluates to true) ...".
- The section describing `USE` statements should be replaced with one describing header files. The use of multiple precisions simultaneously may be stated as unsupported to simplify the description.
- The need to call particular routines to deallocate memory must be specified while describing any calls that leave memory allocated upon return.
- If the package handles multiple precisions these may be dealt with using a section such as the following:

  > **Package types**
  > The complex versions require C99 support for the `double complex` and `float complex` types.
  > The real versions do not require C99 support.
  > We use the following type definitions in the different versions of the package:
  > *Single precision version*
  > `typedef float pkgtype`
  > *Double precision version*
  > `typedef double pkgtype`
  > *Complex version*
  > `typedef float complex pkgtype`
  > *Double complex version*
  > `typedef double complex pkgtype`
  > Elsewhere, for *single* and *single complex* versions replace `double` with `float`.

- Portability descriptions should be updated to reflect dependence on Fortran 2003's C interoperability. The HSL standard wording for this is "Fortran 2003 subset (F95 + TR15581 + C interoperability)".

- In the specification of subroutines, a function prototype should be specified rather than specifying "a call of the following form". Since this defines the types and intents, these should be removed from argument descriptions.

  For example,

  ```
  call ma97_analyse(check,n,ptr,row,akeep,control,info[,order])
  ```

  becomes

  ```
  void ma97_analyse(int check, int n, const int ptr[], const int row[],
        void **akeep, const struct ma97_control *control, struct ma97_info,
        int order[])
  ```

- Fortran types should be replaced with their C equivalents.
- Optional parameters should be specified instead as "may be `NULL`". The wording "not present" replaced with "is `NULL`" and "present" with "is non-`NULL`". To improve clarity, it may be necessary to reword to avoid double negatives.

Acknowledgments

We would like to thank John Reid and Nick Gould for their comments on drafts of this report.