

Documentation for C use of Messy

A Remark for ACM TOMS, August, 2016

Richard J. Hanson, Fred T. Krogh, and Philip W. Sharp¹

Abstract

This document gives additional information on use of the Fortran code `messy()` described in [2] but called from C routines using the name `cmessy()`. A C test driver illustrates several ways of using the code in that environment. Library versions are created for C that support float and double floating point data formats. Equivalent functionality is maintained for the C and Fortran versions in those precisions. Creating a library usable by C routines requires the use of inter-operable Fortran and C compilers. A preliminary step requires using a Fortran compiler to create a library or equivalent object files.

Contents

1	Introduction	1
2	Getting Started for C Use	2
3	Getting Started, Microsoft Visual Studio	2
4	C Usage	4
4.1	Setting the Number of Structs and Threads	6
4.2	Using Named Output Files in C	6
5	Testing	7
6	With Threads	7
7	Acknowledgments	7

1 Introduction

This interface is provided so that C programmers can take advantage of the features in `messy()`, without a conversion of the Fortran source code. Several ideas in Chap. 10, [1] were used in designing this interface. A “direct” call to `messy()` is made using an intermediate wrapper routine, `callmessy()`.

Although C and Fortran are now inter-operable, some features in Fortran used in `messy()` are currently missing in C:

¹Prepared at Math à la Carte.com by fkrogh@mathalacarte.comk, richard.koolhans@gmail.com, sharp@math.auckland.ac.nz

- C has nothing equivalent to *assumed-shape* arrays in Fortran, Chap. 12.3-6, [3]. But C does have *assumed-size* arrays, Appx B.3, [3].
- C and Fortran do not have inter-operable functionality of *optional* arguments.
- C and Fortran manage equivalent string and character data in different ways.

The design of `messy()` uses assumed-shape arrays and optional arguments to yield an easy-to-use message processor. This functionality is implemented in our C code, but it requires a different look and feel for the routine `cmessy()`.

- a. The C code `cmessy()` calls the Fortran wrapper code, `callmessy()`.
- b. The Fortran intrinsic subprogram `c_f_pointer()` converts the assumed-size C arrays and their sizes to assumed-shape Fortran arrays. These are the actual arguments in calls to `messy()`.
- c. Optional arguments in C are implemented as variable lists of enumerated groups of arguments. These provide the array data type, its rank and size, and a pointer to the data.
- d. Conversion of C string variables to Fortran character variables is handled by passing the lengths of the C strings and constructing an intermediate Fortran character variable to pass to `messy()`.
- e. The set of Fortran wrapper routines with C bindings are found in the Fortran module `cmessycall_m.F90`: `allocate_cmessy_interface`, `deallocate_cmessy_interface`, `get_cmessy_defaults`, `callmessy`, `open_cmessy_files`, and `close_cmessy_files`.

2 Getting Started for C Use

In the accompanying file `messy_doc.pdf` note the subdirectory **C**. Unzipping the distribution file shows its contents. On Linux systems the **make** process will test the **C** installation and compare the inter-language call to `messy()`. For Microsoft Visual Studio users, different testing is required:

3 Getting Started, Microsoft Visual Studio

This section is specific to the use of the Intel C++ compiler packaged within the Microsoft Visual Studio environment. We present the tests with instructions for creating separate Visual Studio *Solutions* using the double precision version of `cmessy()`.

cmessylib Creates a static library `cmessylib.lib` for double precision use.

1. Open a new Visual Studio project. Under the template **Intel (R) Visual C++** choose **Empty Project**. Give the project the name `cmessylib`.
2. Under the *Project Properties* menu item, choose the *Configuration Properties* tab. The choose *General > Configuration Type > Static library*.

3. Under the *Configuration Properties* tab choose *General > Platform Tool Set > Intel C++ Compiler* Close the *Project Properties* window.
4. Add source files to the project. These are the header file **cmessy.h** and the source file **cmessy.c**, found in **C\Src**.
5. Under the *Project Properties* menu item, and the *Configuration Properties > C/C++* tab, enable the preprocessor. Add the additional *Preprocessor Definitions*, **CTYP_=CTYP_d** and **maxt_=2**.
6. Under the *Configuration Properties > C/C++/Language [Intel ...]* tab, enable parallel OpenMP and C99 support.
7. Build the project. This step compiles the code and creates the static library **cmessylib.lib**.
8. The full path name to this file is needed in the testing or application use. Abbreviate this as: **<ctpath> = ...\Projects\cmessylib\Debug**.

ctmessy Calls **cmessy()** in many ways and compares results with the distribution file **C\Drivers\Results\result.d**. If you are assured that **cmessylib.lib** is correctly configured this project step can be skipped.

1. Open a new Visual Studio project. Under the template **Intel (R) Visual C++** choose **Empty Project**. Give the project the name **ctmessy**.
2. Under the *Project Properties* menu item, choose the *Configuration Properties* tab.
3. Under the *Configuration Properties* tab choose *General > Platform Tool Set > Intel C++ Compiler* Close the *Project Properties* window.
4. Add the header files **cmessy.h** and **csample.h**, and the source files **ctmessy.c** and **csample.c** to the project. These are in the distribution directories **C\Src** and **C\Drivers**.
5. Add the distributed data file to the project: **C\Drivers\Results\result.d**. This is used to check results.
6. Under the *Project Properties* menu item, and the *Configuration Properties > C/C++* tab, enable the preprocessor. Add the additional *Preprocessor Definitions*, **CTYP_=CTYP_d**, **maxt_=1** and **MSWin**.
7. Under the *Configuration Properties > C/C++/Language [Intel ...]* tab, enable C99 support.
8. Under the *Project Properties* set the path name for the linker to find the libraries containing **messy()** and **cmessy()**: *Linker > General > Additional Library Directories* are set to **<ctpath>**. The second path name is for **<tpath>**, as noted in **messy_doc.pdf**.
9. Under the *Project Properties* set the path name for the linker to use the libraries containing **messy()** and **cmessy()**: *Linker > Input > Additional Dependencies* is set to **dmessylib.lib; cmessylib.lib**.
10. Build and execute **ctmessy**. The results are written to the file **...Results/result.d**.

1. Open a second Visual Studio project within the *Solution*, *ctmessy*. Under the template Intel (R) Visual Fortran choose **Console Application** and next choose **Empty Project**. Name the project *checkctmessy*.
2. Add the source file *checkctmessy.c* to the *checkctmessy* project. This is in the distribution directory *C\Drivers\MSWin*.
3. Add the MS DOS batch file *checkctmessy.bat* to the *checkctmessy* project as a Resource. This is in the distribution directory *C\Drivers\MSWin*.
4. Build and execute *checkctmessy*. The results in file *...Results\result.d* are compared with those of *result.d*.

4 C Usage

The usage in C is similar to that in Fortran, but the specification of optional arguments uses the C *variable argument list* macros. The Text argument parameters have the identical meanings as with Fortran calls to *messy()*.

```
#include "cmessy.h" // <This file is in the package.>
struct cmessy_ty e; // < Your name could be different.>
<Declare other types that you need.>
allocate_cmessy_interface (int maxcstructs, int maxcthreads);
get_cmessy_defaults(&e); // In this order!
<Make changes desired in components of e.>
cmessy(&e, "Text defining what you want, as in a call to messy.",
<optional arguments (see below)>, 0); //Trailing 0 says the list is complete.
```

Listing 1: Sample use of *cmessy()* to print four digits of two complex numbers

```
#include "cmessy.h"
int main(void)
{struct cmessy_ty e;
  allocate_cmessy_interface(1,1); get_cmessy_defaults(&e);
  ck w, z, r[2];
  w=I; z=csqrt(w); r[0]=w; r[1]=z;
  cmessy(&e, '$D4The square-root of the complex number $ZR = $ZR',
    m_zdat, 2, r, 0);
  // The square-root of the complex number (0.,1.000) = (.7071,.7071)
}
```

The type *cmessy_ty* has a C binding and the following public components. The indicated defaults are set by the call *get_cmessy_defaults()*.

```
character(c_char) ::  ename[32]="Undefined" ! Name printed in error messages.
integer(c_int)  ::  fpprec = numdig ! Default for floating point precision
integer(c_int)  ::  kdf = numdig ! Current default real precision.
integer(c_int)  ::  line_len = 128 ! Default for line length
integer(c_int)  ::  munit = OUTPUT_UNIT ! Message unit number
integer(c_int)  ::  eunit = OUTPUT_UNIT
```

```

integer(c_int) :: maxerr = 0 ! Max value of 1000 * (10*stop + print) + |error index|
integer(c_int) :: lstop = 3 ! Stop indexes ≤ this don't stop
integer(c_int) :: lprint = 3 ! Print indexes ≤ this don't print
integer(c_int) :: errcnt = 0 ! Count of the number of error messages, incremented
!                               by 1000000 for internal errors inside messy.
integer(c_int) :: dblev = 3 ! If 0, an immediate return is made (unless text
!                               starts with "$E"), else a $K<integer> will behave
!                               as if reaching the end of text if <integer> is > dblev.
integer(c_int) :: cinit = 1234565 ! Note if get_cmessy_defaults(&e) was called
integer(c_int) :: cstruct ! Index of internal copy of messy_ty used to call messy

```

The next table shows the correspondence between how things look in a call to the Fortran routine `messy()` compared to calling the C routine `cmessy()`. We use “dim” for the dimension of a rank-1 array, “rdim” and “cdim” for the row and column dimensions of a rank-2 array.

Fortran Args.	C Arg. Groups
idat=idat	m_idat , dim, idat
rdat=rdat	m_rdat , dim, rdat
zdat=zdat	m_zdat , dim, zdat
imat=idat	m_imat , rdim, cdim, idat
rmat=rdat	m_rmat , rdim, cdim, rdat
zmat=zdat	m_zmat , rdim, cdim, zdat
ix=ix	m_ix , dim, ix
ptext=ptext	m_ptext , ptext

Note that **0** must be the last argument after all groups. This signals the end of the list of variable arguments. The leading integer flag in each group is defined in a sequential enumeration, located in the header file `cmessy.h`. The value of the flag indicates the type, rank and size of the array that is to be printed by `messy()`. The C integer dimensions are passed to Fortran by value. To print a single value pass it by pointer reference.

Listing 2: Call to `cmessy()` that prints eight digits of a matrix with 0-based subscripts

```

#include "cmessy.h"
int main(void)
{struct cmessy_ty e;
allocate_cmessy_interface(1,1); get_cmessy_defaults(&e);
rk rmatrix[2][3]={ {1,2,3},{4,5,6}};
for(int i=0;i<2;i++){
    for(int j=0;j<3;j++){rmatrix[i][j]=sqrt(rmatrix[i][j]);}}
cmessy(&e,"$D8rmatrix:$N$A$O0$O0$O",m_rmat,2,3,rmatrix,0);
cmessy(&e,"rmatrix[0][1]=$R",m_rdat,1,&rmatrix[0][1],0);
/*
rmatrix:
      Col 0      Col 1      Col 2
Row 0 1.0000000 1.7320508 2.2360680
Row 1 1.4142136 2.0000000 2.4494897
rmatrix[0][1]=1.4142136 */
}

```

The call to the Fortran routines `allocate_cmessy_interface` and `get_cmessy_defaults`, in this order, are *required before* any call to `cmessy()`. These calls set the initial state in the wrapper code `callmessy()` and return initial values for the C structure `e`.

4.1 Setting the Number of Structs and Threads

Arrays in the wrapper routine `callmessy()` are allocated with an initial call

```
allocate_cmessy_interface (int maxcstructs, int maxcthreads);
```

The first argument `maxcstructs` is an upper bound for the number of different copies of the structure `cmessy_ty` used within an application. The second argument `maxcthreads` is an upper bound on the number of OpenMP threads created in parallel sections or loops. If there are no calls to `cmessy()` within OpenMP parallel sections, use `maxcthreads=1`.

To reset the number of structures and threads, deallocate and then allocate again with the new numbers:

```
deallocate_cmessy_interface (void);  
allocate_cmessy_interface (int new_maxcstructs, int new_maxcthreads);
```

4.2 Using Named Output Files in C

Associated with each structure `e` that is used in calls to `cmessy()`, one can designate output to be placed in a named file. Note that this cannot be done directly in the C code, as it has no access to the Fortran I/O libraries. Following the call to `get_cmessy_defaults(&e)`, and prior to `cmessy()`, call the routine,

```
open_cmessy_files(&e, Task, File).
```

Task Defines what output is associated with **File**: 1 for regular messages, 2 for error messages, 3 for both, and 0 to set the error unit to the Fortran `ERROR_UNIT` (**File** not used).

File A character string, giving the desired name. This can be a full path name.

All `cmessy()` files opened may be closed with

```
close_cmessy_files(&e).
```

The facility for writing messages to a named file may be useful in a parallel computing environment where only one node is able to process I/O. Each processor assigns different units to itself, and messages for each processor would go to different units. The one node capable of doing I/O could pick up the messages from each of these output units without messages from different processes getting mixed together in a confusing way.

5 Testing

The code has been tested on the following systems.

Gentoo Linux: gfortran/gcc 4.9.2, 5.1.0, 5.4.0, 6.1.0

Ubuntu Linux: gfortran/gcc 5.3.1 and 5.4.0

Ubuntu Linux: NAG 6.1 gcc 5.3.1

Red Hat Enterprise Linux: ifort 13.0.1

6 With Threads

We intended to provide a C testing code for using OpenMP threads. Its design was similar to the test code `thrdtmessy`, documented in `messy_doc.pdf`. We have had difficulty in getting this part of the project to work in a portable way, and thus this is not included in the current code.

7 Acknowledgments

The authors are indebted to W. Van Snyder for answering many questions about the Fortran standard, and for suggestions on alternative ways of doing things to take advantage of what the latest versions of Fortran have to offer. Tim Hopkins has helped by locating problems in using the NAG compiler in earlier versions.

References

- [1] HANSON, R. J., AND HOPKINS, T. *Numerical Computing with Modern Fortran*. SIAM Publications, Philadelphia, PA, USA, 2013.
- [2] KROGH, F. T. A Fortran message processor. *ACM Trans. Math. Softw.* 40, 2 (Feb 2014), Article 15.
- [3] METCALF, M., REID, J., AND COHEN, M. *Modern Fortran Explained*. Oxford University Press, Oxford, UK., 2011.