Messy Documentation

Revised for ACM TOMS, August, 2016

Fred T. Krogh, Richard J. Hanson, and Philip W. Sharp[1]

Abstract

This document gives more information on the testing and use of the code described in [1] plus many more details on the packaging. Two test drivers illustrate the use of several ways of using the Fortran code. The first code shows use with a single thread and the second shows use with two threads. By implementing a generic interface, a single library version can support single, double and quadruple floating point data formats.

Contents

1 Introduction

We have two goals here. One is to describe how to set up and use the subroutine `messy()` to output nicely formatted text and error messages. The other is to show by example how libraries can be set up to support different precisions, for use with or without threads, and in different computing environments. Ideas related to those in [2] were used. By following the example here one can create libraries that can execute with different precisions.

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

We believe this complexity pays off with the flexibility provided in later use of the software. Although technically not part of the Fortran standard, the C-style preprocessor built into the compilers is used. The preprocessor directives reduce the number of source files that would be required otherwise.

The test program `tmessy` illustrates all of the different types of output supported by `messy()`. For complicated use we recommend looking at the relevant example there. An additional test program illustrates support for OpenMP threads in Fortran. Without this support one may find output from within parallel sections merged in a way that makes it difficult to interpret which thread is doing what. In the test program, `thrdtmessy`, numerical output and error messages from different threads are written to different named files.

Due to a serious illness of the second author, the testing for Microsoft Windows and for Cray compilers has not been done. We have left in text describing what used to work, but changes will be needed beyond what is provided here. `Makefile`, possibly with some editing, should work under Windows using Cygwin[2].

2 Getting Started

Unzipping the distributed file `936.zip` will create the following directory structure. The subdirectory for **C** is referenced in the companion file `cmessy_doc.pdf`. It can be ignored here.

936
- `Makefile` ! For use on Linux systems or Windows systems with *Cygwin*.
- `936.head` ! Part of ACM TOMS/CALGO packaging
- `936.README` ! Quick instructions to get started.
- **Doc**
  - `messy_doc.pdf` ! This Document
  - `cmessy_doc.pdf`
  - `messy_doc.tex`, `messy_doc.bib` ! In case you would like to make changes.
- **Fortran90**
  - `Makefile` ! Used for making the Fortran part of the package on Linux.
  - **Src**
    - `messy_m.F90` ! The main part of the package.
    - `precision_m.F90` ! For choosing the floating point precision used in the codes.
  - **Drivers**
    - `tmessy.F90` ! Shows most of the features of the code
    - `sample_m.F90` ! Shows how library packaging can be used
    - `thrdtmessy.F90` ! Illustrates the use of threads
    - **Results** ! This directory has expected results from running these codes
      - `result.s, result.d, result.q` ! Results for different precisions
      - `odd1.?, mes0.?, err0.?` ! From testing threads where ? is `s, d`, or `q`.
  - **MSWin** ! For checks using Windows Visual Studio
    - `checktmessy.f90, checktmessy.bat, checkthrdtmessy.bat`

[2]For example see `https://www.cygwin.com/`

C

- `Makefile` ! Used for making the C part of the package on Linux.
- **Src**
  - `cmessy.c` ! As for `messy()`, but called from C with a different interface.
  - `cmessy.h` ! An include file needed when compiling C.
  - `cmessycall_m.F90` ! Translates the C calls to one made to `messy()`.
- **Drivers**
  - `csample.c` ! Shows how library packaging can be used for C.
  - `csample.h` ! Include file needed for `csample.c`.
  - `ctmessy.c` ! Driver illustrating the use of `cmessy` similar to `tmessy` for Fortran.
  - **Results** ! This directory has expected results from running `ctmessy.c`
    - `cresult.s, cresult.d` ! Results for two precisions
- **MSWin** ! For checks using Windows Visual Studio
  - `checktmessy.f90`, `checktmessy.bat`

2.1 Using the make file

In many cases one can define compilers on the command line with "make FC=...CC=...", and never touch the make file. But if your compiler is not covered or you have some special needs, you should start by editing `Makefile`. Comments in the make file describe how to set up the number of threads, the precisions you would like created and tested, and the directory structure used. In addition there are variables to handle the fact that different computing environments, use different ways to express the same option. There are if statements that set variables depending on the compiler used.

Running make, will run the make file in the `Fortran90` subdirectory which will create code in the `Libs` directory and results in `Fortran90/Drivers/NewResults`. These results are compared with results generated on a Linux machine running gfortran and gcc. If C code is generated, after the comparison of the Fortran results are printed, the make file in the `C` subdirectory is run. After generating results for the C code, the results are compared with those generated on a Linux machine running gfortran and gcc.

If you don't use make, you will need to understand variables in the `Makefile`. The comments there describe all the variables used. Substitutions, also defined, along with loops, allow the make file to generate all the precisions required.

In order to generate the code for the various precisions, variables are defined which are passed into the Fortran compiler. You can find there variables by looking for "-D" in the options used by the compilers.

Some variables in `Makefile` are used to define variables for the preprocessor. Those preprocessor variables appear in the source code. The variables used are given below.

Although this code has not been checked out on windows, it was at one time. Some drivers use the preprocessor variable "MSWIN", to introduce alternate code needed for the Windows environment. So in that environment one should ensure that this preprocessor variable is defined, prior to compilation.

The targets that can be specified when using make are

make clean Gets things back to the original state.

make Make directories as needed, compile codes, run them, and compare results with what is provided in the distribution.

make Doc/messy_doc.pdf Useful if you make changes this file.

make replace_orig Useful if you want to try more threads, and would like the output for what is saved for the distribution to include more output. You might also use this if you see minor differences between your results and ours, and would prefer not to see them again. You should be sure that the results you are getting are good ones before using this.

3 Getting Started, Microsoft Visual Studio

What is described here worked at one time, but probably needs some massaging to get this to work. This was working prior to having messy available as generic software.

This section is specific to use of the Intel compiler packaged within the Microsoft Visual Studio environment. Since MS DOS does not provide a native *make* tool, we present the tests with instructions for creating separate Visual Studio *Solutions* using the double precision version of `messy()`.

dmessylib Creates a static library `dmessylib.lib` and `.mod` files for double precision use.

1. Open a new Visual Studio project. Under the template `Intel (R) Visual Fortran` choose `Library` and next choose `Static Library`. Name the project `dmessylib`.
2. Add source files to the project. These are `messy_m.F90` and `precision_m.F90`, found in the distribution directory `Fortran90\Src`. For inter-language use with **C**, add the module `cmessycall_m.F90`, found in `C\Src`.
3. Under the *Project Properties* tab, enable the *Fortran > Preprocessor* and add the directives `kind_=kindd`[3] and `numt_=i` where is is any integer > 1. Next enable the parallel processing of OpenMP directives under *Fortran > Language*.
4. Build the project. This step compiles the codes, creates `.mod` and `.o` files, and the static library `dmessylib.lib`.
5. The full path name to these files is needed in the testing. Abbreviate this as: `<tpath> = ...\Projects\dmessylib\dmessylib\Debug`.

tmessy Calls `messy()` in many ways and compares results with the file `Fortran90\Drivers\Results\dresult`. This step can be skipped if you are assured that `dmessylib.lib` is built correctly. There are two *projects* within this solution, even though it is called a project below.

[3]Chooses `double precision` for floating point data precision.

1. Open a new Visual Studio project. Under the template `Intel (R) Visual Fortran` choose `Console Application` and next choose `Empty Project`. Name the project `tmessy`.
2. Add the source files `tmessy.F90` and `sample_m.F90` to the project. These are in the distribution directory `Fortran90\Drivers`.
3. Under the *Project Properties* enable the *Fortran > Preprocessor* and add the directives `MSWin` and `plet_='d'`. Next define the path name for the `.mod` files: Set *Fortran > Preprocessor > Additional Include Directories* as `<tpath>`. This step is for using the `.mod` files for modules `messy_m` and `precision_m` during compilation of `tmessy.F90`.
4. Under the *Project Properties* set the path name for the linker to find the library containing `messy`: *Linker > General > Additional Library Directories* is set to `<tpath>`.
5. Under the *Project Properties* set the path name for the linker to use the library containing `messy()`: *Linker >Input > Additional Dependencies* is set to `dmessylib.lib`.
6. Build the project and execute `tmessy`. The results are written to the file `newresult.d`.

1. Open a second Visual Studio project within the *Solution*, `tmessy`. Under the template `Intel (R) Visual Fortran` choose `Console Application` and next choose `Empty Project`. Name the project `checktmessy`.
2. Add the source file `checktmessy.f90` to the `checktmessy` project. This is in the distribution directory `Fortran90\Drivers\MSWin`.
3. Add the file `dresult` to the `checktmessy` project as a Resource. This is in the distribution directory `Fortran90\Drivers\Results`.
4. Add the MS DOS batch file `checktmessy.bat` to the `checktmessy` project as a Resource. This is in the distribution directory `Fortran90\Drivers\MSWin`.
5. Build and execute `checktmessy`. The results in file `newresult.d` are compared with those of `dresult`.

thrdtmessy Calls `messy()` in an OpenMP parallel DO loop with numeric and error messages written to separate named files and compares results with files `mes0.d`, `err0.d` and `odd1.d` found in `Fortran90\Drivers\Results`.

1. Open a new Visual Studio project. Under the template `Intel (R) Visual Fortran` choose `Console Application` and next choose `Empty Project`. Name the project `thrdtmessy`.
2. Add the source file `thrdtmessy.F90` to the project. This is in the distribution directory `Fortran90\Drivers`.
3. Under the *Project Properties* enable the *Fortran > Preprocessor* and add the directives `MSWin`, `plet_='d'`, and `numt_=2`. Next define the path name for the `.mod` files: Set *Fortran > Preprocessor > Additional Include Directories* as `<tpath>`.

This step is for using the `.mod` files for modules `messy_m` and `precision_m` during compilation of `thrdtmessy.F90`. Enable the parallel processing of OpenMP directives under *Fortran > Language*.

4. Under the *Project Properties* set the path name for the linker to find the library containing `messy()`: *Linker > General > Additional Library Directories* is set to `<tpath>`.
5. Under the *Project Properties* set the path name for the linker to use the library containing `messy()`: *Linker >Input > Additional Dependencies* is set to `dmessylib.lib`.
6. Add the MS DOS batch file `checkthrdtmessy.bat` to the `thrdtmessy` project as a Resource. This is in the distribution directory `Fortran90\Drivers\MSWin`.
7. Add the data files `mes0.d`, `err0.d` and `odd1.d` from the algorithm distribution as a Resource.
8. Build and execute `thrdtmessy`. The results in subdirectory `...\NewResults`, `mes0.d`, `err0.d` and `odd1.d`, are compared with those from the the algorithm distribution.

4 General Fortran Usage

To use the `messy()` software, arrange for access to `.mod` and a library or `.o` files for modules `messy_m` and `precision_m`.

use `messy_m`, only : messy, messy_ty, rk
type(`messy_ty`) :: `e` ! Your name could be different.
<Declare other types that you need, and/or change public components of `messy_ty`.>
. . .
call `messy`(`e`,'The text that defines what you want.', other arguments as needed)

Listing 1: Sample use of `messy()` to print four digits of two complex numbers

```
use messy_m, only: messy, messy_ty, rk
    type(messy_ty) :: e
    complex(rk) :: w=(0.0,1.0)
     call messy(e,'$D4The square-root of the complex number $ZR = $ZR.', &
      zdat=[w,sqrt(w)])
    end
! The square-root of the complex number (0.,1.000) = (.7071,.7071).
```

The precision used by messy for real and complex numbers is specified by the value of the named constant (or parameter) `rk`. This value is defined at the start of `messy_m.F90` with the statement
"`integer, parameter :: numdig = ceiling(-log10(epsilon(1.0_rk)))`". The type `messy_ty` has the following public components with the specified default and initial values.

`character (len=32) :: ename="Undefined"` ! Name printed in error messages.
`integer :: fpprec = numdig` ! Default for floating point precision

```
integer ::  kdf = numdig ! Current default real precision.
integer ::  line_len = 128 ! Default for line length
integer ::  munit = OUTPUT_UNIT ! Message unit number
integer ::  eunit = OUTPUT_UNIT ! ERROR_UNIT mixes up output with piping
integer ::  maxerr = 0 ! Max value of 1000 * (10*stop + print) + —error index—
integer ::  lstop = 3 ! Stop indexes ≤ this don't stop
integer ::  lprint = 3 ! Print indexes ≤ this don't print
integer ::  errcnt = 0 ! Count of the number of error messages, incremented
!                         by 1000000 for internal errors inside messy().
integer ::  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.
```

In addition this data type contains space for an additional 13 integers used internally.

The parameters that can be passed into `messy()` are as follows:

```
  subroutine messy(e, text, idat, rdat, imat, rmat, zdat, zmat, ix, ptext)
    type(messy_ty), intent(inout) :: e    !Required
    character(len=*), intent(in) :: text  !Required
    character(len=*), optional, intent(in) :: ptext !The rest are optional
    integer, optional, intent(in) :: idat(:), imat(:,:), ix(:)
    real(rk), optional, intent(in) :: rdat(:), rmat(:,:)
    complex(rk), optional, intent(in) :: zdat(:), zmat(:,:)
```

The subroutine messy scans the character string supplied by the text argument for actions to perform. When it sees a dollar sign, it interprets the next few letters to format and output the data values in the idat, imat, rdat, . . . zmat arrays, ptext, or to adjust other parameters connected with formatting. The text not interpreted as dollar sign commands is text to be printed on the specified output unit. When output from `idat`, `rdat` or `zdat` is requested the first number printed is the first location in the array and further requests are always from the location just after the last one printed.

Before a call to `messy`, you can change any of the default values defined in `messy_ty` above, but there is another mechanism that can be used to change some of these values on a more temporary basis, in the `text` argument using a $ followed by letters as described below. The $ which serves as an escape character can be changed by changing the value of the parameter `sc` in `messy_m.F90`. Only certain characters are allowed after a $ and the actions associated with these are listed below. In describing some of these we use "[· · ·]" to indicate something is optional, "[integer]" indicates an optional integer, "text" is used for any text, and # is used for a single decimal digit.

A Print a real matrix from `rmat`. If column and row headings have not been changed using $O (see below), column headings have the form Col nnn, and Row headings have the form Row nnn, where nnn is the index of the column or row. AT results in the transposed matrix being printed.

B Break, restores the defaults in `e%fpprec`, `e%line_len`, `e%munit`, ends a table if such is active and returns.

C Continue. $C, can be used to end an integer when followed by a digit. When this is the last thing in text, and we are processing an error message, the error message is not ended at this point, but is continued on the next call. This is useful for output of multiple vectors in an error message.

D $D[integer] is used to specify a temporary number of significant digits for floating point output. If the integer is missing it restores the default, if the integer is ≤ 0, then this specifies the negative of the number of digits that must follow the decimal point and that no exponent is to be used in the output (and thus large numbers will use a lot of space). When this is set on one call, it is used for later calls unless a $B is used to restore the value in `e%fpprec`.

E Start an error message, see Section 5. The next two characters are digits, the first gives the stop index, and the next the print index. If the stop index is 0, it is not treated as an error, but can be used to limit printing of other messages. If this is an error message the index of the error, defined by the package generating the error message, is in `ix(1)` if it is present (useful if some integer vector is part of the error message), and else is in `idat(1)`. (And if `idat` is not present, then a 0 is printed for the error index.) Note that after any print from `idat` the next integer printed from `idat` will come from the next location. If the stop index is 0, print will still come out on the error unit.

F Define an alternative format for integer or floating point output. The $F is followed by a “D”, or one of the letters: “IFE” followed by “digits.digits”, where the “. digits” is optional for “I”. The “D” case is primarily intended for use with tables, see item “H” below, and gives a Fortran “ES” format to print the number of significant digits that would ordinarily be given by e%kdf (or was last set by $D) and with a width sufficient to give one space at the start of the number. The “E” cases are converted to Fortran’s scientific edit descriptor “ES”. This also formats the next value according to the format item, either for the I case, the next integer value from the idat, for the real cases, the next real value from rmat, or for the $ZF case, the next complex value from zdat. In the complex case, the format provided is used for both the real and imaginary parts unless the format is terminated with a comma (”,”) in which case the format for the imaginary part follows. These formats are not saved from one call to the next.

G Output the next real from `rdat` using the last real format specified by $F above.

H In the middle of text, $H specifies a preferred place for breaking a line. If this is before the start of text, it specifies the start of a table. By a table we mean text that is arranged in columns where the caller indicates what is to go in each column. If there are more columns than will fit on a line, text that does not fit will be saved in a scratch file. In most cases the end of the table will be indicated with text containing nothing but a break specified by $B. The user is responsible for setting up headings, and formatting the following lines so things line up as desired. The following lines will make use of formatting available through the $F command above. The heading line has the following form: “$H—nctext1—nctext2— ⋯ —nctextk—”, where “—” is any character the user wants to use, different than anything else in the current text string; n is a number giving the number of spaces needed for the column (if 0 or missing the column is later formatted with “$FD”, see “F” above); and c is a C, R, or L indicating how the following text is to be justified in the column. The test program `tmessy.F90` gives an example for setting up tables. One can have at most `lenbuf`−50 characters in a heading where `lenbuf` is a parameter in `messy_m.F90`, currently = 256. Note that one can make up headings, and output table entries with a size that depends on the precision, by using "$FD", and setting e%kdf to a value that depends on the precision (perhaps using "$D"). **WARNING**, nothing but the body of the table can be output by `messy()` until the "$B" that ends the table.

I Print the next integer in `idat`, and continue.

J As for I above, except use the last integer format defined by a "$F", see above.

K This is followed by an integer (assumed 0 with no digits given). If that integer is > `e%dblev`, then actions are as if `text` has ended at this point. If `e%dblev` is 0, then an immediate return is made unless `text` starts with "$E", or we are already processing an error message. The $K<integer> is ignored when processing an error message. If this feature is used, then the smaller the integer following the $K, the more likely the following text is to print. The larger the number in `e%dblev`, the more likely text after a "$K" is to print.

L Followed by an integer giving a new line length. Internally the number specified is replaced as necessary to get it in the interval [40, lenbuf−50] (parameter lenbuf in the code is now 256).

M Print a matrix from `imat`. Headings are as for $A. $M$T results in the transposed matrix being printed.

N Start a new line, and continue.

O Define starting indexes for vector or matrix output and for the matrix case, alternative formats for output of column and row labels. The $O may only appear immediately after a $A, $M, $V, or $W or after a $T following one of these. Following the $O is an optional integer giving the starting index value (which may be negative). No integer gives the default, which is 1. In the matrix case one can indicate the text to output for the column and row headings, and whether and where the index should be printed. A $O following a $A or $M, is followed by text which specifies the desired result for columns, a $O, then specifications for rows, and then a terminating $O. Following the optional integer giving the first index, one may have nothing, or a specification of the number of characters in the text for the headings in each column, and where indexes are to be printed followed by the actual text used for headings. "<#" means print the index first with # characters in the text for each column heading (all must be the same). ">#" is the same but the heading text follows the index. " #" gives no index, but just # characters of heading. "|##" has the first # characters of heading text then the index, and then the following # characters of heading text. The sum of the digits in the ## must not exceed 9.

If the text is the same for all columns (or rows) then `text` contains L characters, otherwise it contains some multiple of L characters, and different text is used for each heading until running out of text in which case the last given is repeated. One can get the default actions by not using the $O, with $O$O$O, with $O>4Col $O>4Row $O, $O>4Col $O$O, or $O$O>4Row $O. If column headings were to be Earth, Air, Fire, and Water and row labels were to have the form "Case <index>:", this would be $O 5Earth Air FireWater$O|51Case :$O. Column headings are centered over the data for that column, unless the column headings are wider than needed by the data in which case the data is right justified with the heading.

P Print the text from ptext at this point. Useful if you call an internal subroutine with a common message except for one small bit of text.

Q This is used to print integers as bit, octal, or hexadecimal digit strings. If ix is not defined it is assumed that the digit string is in a single integer, and that leading 0's are not to be printed. If ix is present then |ix(1)| defines the number of digits, and if ix(1) > 0, then this gives the number of digits and all are printed, otherwise it is assumed that leading 0's should not be printed, and the digit string consists of at least one integer. `idat` contains the integers to be printed as digit strings much like when `idat` is used for output of integers or integer vectors, with the difference that a single digit string may require more than 1 word. Following the $Q must be a B, O or a Z, indicating that one wants the digits output in binary, octal, or hexadecimal. Following this must be an I or a V indicating that one wants just a single digit string output, or would like to output the contents of `idat` as a vector of digit strings. Since Fortran does not have a bit data type, the high order bit of the integers is ignored, thus if one has 32 bit integers, each integer string only uses the lower 31 bits.

R Print the next real entry in `rdat`, and continue.

S Print the real sparse vector, with row/column indexes in idat and values in rdat. Text at the start of the line containing the $S is printed at the start of each line.

T Tab to the column that is a multiple of that set with $<integer>T, see 0–9 below. If used immediately after a $A or $M, $T causes the transpose of a matrix to be printed.

U Set the output unit to the following integer. This output unit will be used until the next $B.

V Print a floating point vector from `rdat`. If other numbers have been printed from `rdat`, then the vector starts with the first unprinted number.

W As for $V, but for integers from `idat`.

X Print an integer from `ix`. $X prints `ix`(1), $X<integer> prints ix(<integer>).

Y,y If you would like to repeat what is in part of text ix(1) times, you can do this with $Y, and then indicate the end of that block with $y. After processing the text in $Y...$y ix(1) times, text is then processed in the usual way. This was introduced primarily for tables ($H) so that tables could have a variable number of columns. When used for column headings, the $Y and the $y should both be followed by the special character used to delimit column headings.

Z This is used for complex data from `zdat` or `zmat`. It must be followed immediately by A, F, G, R, or V, with these giving the same result as these give in the real case when preceded by a "$".

0–9 Starts an <integer> and then either a F, G, J, T, or a blank will repeat the $F, $G, or $J action that number of times, or set the column multiple for tab stops, or output that many blanks.

$ A single $ is output.

Default Anything else gives an error message and returns.

Examples illustrating all the features above are in `tmessy.F90`. Here we just give a simple example. Suppose you have a real matrix "rmatrix" that you would like to print with 8 significant digits, and with a line length of 100. To have "rmatrix:" printed along with the matrix, you could use this fragment:

Listing 2: Use of `messy()` to print eight digits of a real matrix

```
 use messy_m, only: messy, messy_ty, rk
    type(messy_ty) :: e
    real(rk) :: rmatrix(2,3)=real(reshape([1,2,3,4,5,6],[2,3]),rk)
    call messy(e,"$D8$L100rmatrix:$A$B", rmat=sqrt(rmatrix))
    end
! rmatrix:
!            Col 1      Col 2      Col 3
! Row 1  1.0000000  1.7320508  2.2360680
! Row 2  1.4142136  2.0000000  2.4494897
```

You could just accept the default digits (full precision), and the default line length (128), and you can always make permanent changes to the defaults in `messy_ty`. Leave off the `$B` if you want the default digits and line length to remain changed.

One might guess that if you pass in a full array, `messy()` would adjust indexes printed to match those of the array. But Fortran does not work this way. Fortran always passes arrays into `messy()` with a lower bound of 1. If you want indexes to match those of an array that does not have a lower bound of 1, use the above $O feature.

5 Error Messages

When it comes to printing or stopping on an error message there is no single right answer. Sometimes the user would like the code to print and not stop, or print and stop, or do neither depending on the message, and on the context in which the code is being run. And it may be that the user would like the error messages and other messages to go to separate files.

For the person writing the code which is printing the error message, they should have set `e%ename` to the name they use to identify the package as part of the initialization. For any particular message they can indicate how important they think it is for this message to stop and to print. The text for an error message must start with `$Esp`, where `s` is a digit indicating how important it is to stop on completion of printing this error message, and `p` is a digit indicating how important it is that the message print. One should only use a 9 for `s` if they have given a flag to the user indicating that without appropriate action on their part the program will be stopped. For errors in passing arguments to `messy()`, the routine makes a recursive call to itself with a stop index of 4, and a print index of 8. One can avoid a stop on these internal errors by setting `e%lstop=4`. One must have `p` $\geq$ s and `p`=9 only if `s` is 9. By default a line of $'s is printed before and after an error message. For different actions one can change the parameter `errchar` in `messy_m.F90`. One is encouraged to use multiple calls to output all possible information of potential interest concerning the error.

All but the last call for an error message should end with a $C, and only the first has the starting $E.

For the person using a package that processes errors they should have access to the data in “`e`”, in order to change the rules for stopping, printing, line lengths, output units, etc. Note that `e%maxerr` will contain the index of the stop index and print index for the error that seemed most serious, and `errcnt` will have a count of the total number of error messages.

6 With Multiple Levels of Routines

Suppose a user is calling a boundary value solver, `bsolve`, and an error occurs in a linear optimizer, `losolve` which in turn was called by a nonlinear optimizer, `nlsolve`. It is likely that an error message from `losolve`, will not give the user the clarity needed to correct a problem, since he is only aware of the interface to `bsolve`.

But if `bsolve` contains a derived type `bsolve_ty` which contains inside it a variable of derived type `nlsolve_ty`, and that data type contains inside it a type of `losolve_ty`, and each of these derived type also include a variable of derived type `messy_ty`, then upon getting the error from `losolve`, `nlsolve` can translate that error into an error that makes sense from the view of the nonlinear solver, and as that error is passed up the line, `bsolve` can finally output an error message in a form that is more likely to be understandable to the user, since `bsolve`, has access to error information all the way down the call tree.

The facility for writing messages to a named file is particularly useful in a parallel computing environment. Each processor assigns different output units to itself, and messages for each processor would go to these units. The one node capable of displaying I/O could pick up the messages from each of these output units, preventing individual records from different processes getting disordered in a confusing way.

7 Threads

If you want to use threads, you should pay attention to the −`D` options used in `Makefile` for threads, and of course examine `thrdtmessy.F90` to understand better what is needed. Variable `T`, defined in `Makefile` is required for threads.

In the comments of `thrdtmessy.F90` you will find five modes for scheduling threads: STATIC, DYNAMIC, GUIDED, RUNTIME, or AUTO. There is no guarantee that the output files from different environments will give the same outputs, but all output files as a group should contain all that is written.

A programmer can declare variables of type `messy_ty` to be OpenMP *threadprivate* or they can allocate an array that corresponds to the maximum number of threads used. Whatever the choice, we recommend using separate named files for output when there is any chance of confusion with numeric output or error messages.

8 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

CentOS release 6.4 ifort 13.0.1 20121010, icc 13.0.1 20121010

Testing by a referee has revealed that on some older systems, the make files will need changes.

9 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.

The authors wish to acknowledge the contribution of the NeSI high-performance computing facilities at the University of Auckland and the staff at NeSI and Centre for eResearch. New Zealand's national facilities are provided by the New Zealand eScience Infrastructure (NeSI) and funded jointly by NeSI's collaborator institutions and through the Ministry of Business, Innovation and Employment's Infrastructure programme.
URL `http://www.nesi.org.nz`.

References

[1] Krogh, F. T. A Fortran message processor. *ACM Trans. Math. Softw. 40*, 2 (Feb 2014), Article 15.

[2] Van Snyder, W. Poor man's templates in Fortran 90. *SIGPLAN Fortran Forum 16*, 3 (1997), 11.