A

Messy Documentation

Prepared for ACM TOMS, January, 2013 Fred T. Krogh[1]

1. SETUP

To use this software,

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)

The real kind `rk` is defined in `precision_m.f90` and defines the type of real and complex variables passed to messy. The standard default for the number of digits to print for real and complex number is defined at the start of `messy_m.f90` with "`integer, parameter :: numdig = ceiling(−log10(epsilon(1.0_rk)))`". The type `messy_ty` has the following public components.

```
integer :: fpprec = numdig ! Default for floating point precision
integer :: kdf = numdig ! Current default real precision.
integer :: line_len = 128 ! Default 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) + |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
integer :: dblev = 3  ! See item K below for a full description.
character (len=32) :: ename = '?' ! Package name to print with error messages
character (len=2) :: echars = '$8'! First character is the error message separator.
! '0' give no separator, ' ' gives a blank line. Second character is the stop level for
! errors in the use of messy.
```

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
  character *(len=*), intent(in) :: text
  character *(len=*), optional, intent(in) :: ptext
  integer, optional, intent(in) :: idat(:), imat(:,:), ix(:)
  real(rk), optional, intent(in) :: rdat(:), rmat(:,:)
  complex(rk), optional, intent(in) :: zdat(;), zmat(:,:)
```

Before a call to messy, you are free to 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 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.

When output form either idat or rdat is requested the first number printed is printed from the first location in the array and further requests are always from the location just after the last one printed.

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.

B. Break, restores all defaults and returns.

[1]Prepared at Math à la Carte.com by fkrogh@mathalacarete.com

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

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). The usual default is always restored after a $B.

E. Start an error message, see Section 2. 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 alternate format for integer or floating point output. The $F is followed by one of the letters: IFE, and then "digits.digits", where the ". digits" is optional for "I". The "E" cases are converted to Fortran's scientific edit descriptor "ES". This also does the type of output specified by the format. (Either the next integer from `idat` in the "I" case, the next real from `rdat` for real formats, or in the $ZF case, the next complex number from `zdat`. In the complex case the format provided is used for both the real and imaginary part unless the format is terminated with a ",", 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 the first thing in text it specifies the start of a table. By a table we mean text that is arranged in columns where the caller indicates where line breaks may occur. If there are more columns than will fit on a line, text that does not fit will be saved in a scratch file, and that text will be output when the end of the table is indicated with a $B. In most cases the end will be indicated with text containing nothing but the $B. The user is responsible for setting up headings, and formatting the following lines so things line up as desired. Using $F (or $<integer> followed by F) should be used for this purpose. The text from the first $H up to the next $H is text that for every row will be repeated when data is needed from the scratch file. Later $H's signal where it is acceptable for a line to be broken. The text for headings should extend to exactly the same distance as is required by later lines. The test program `tmessy.f90` gives an example for setting up tables, including suggestions on how to match the heading line with the following lines. One can have at most lenbuf−50 characters in a line where lenbuf is a parameter in `messy_m.f90`, currently = 256.

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` had 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 gives a new line length. Internally the number specified is replaced as necessary to get it in the interval [40, lenbuf−50] (parameter lenbuf in code is now 256).

M. Print a matrix from `imat`. Headings are as for $A.

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. 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 headings and where indexes are to be printed followed by the actual text used for headings. "<#" means print the index first with # digits in the heading. ">#" 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 $O, with OO$O, with $O>4Col $O>4Row $O, $O>4Col OO, or OO>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 strings. If ix is not defined it is assumed that the bit 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 bits, and if ix(1) > 0, then this gives the number of bits and all are printed, otherwise it is assumed that leading 0's should not be printed, and the bit string consists of at least one integer. `Idat` contains the bit strings much like when `idat` is used for output of integers or integer vectors, with the difference that a single bit string may require more than 1 word. Following the $Q must be a B, O or a Z, indicating that one wants the bits output in binary octal or hexadecimal. Following this must be an I or a V indicating that one want just a single bit string output, or would like to output the contents of `idat` as a vector of bit 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 holds only 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.

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 the integer in `ix(k)`. The default for k is 1, but k can be changed by following the $X with an integer.

Z. This is used for complex data. 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 ' ' 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.

There are many examples in `tmessy.f90`, but we give a simple example here. 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

```
call messy(e,¨$D8$L100rmatrix:$A$B¨, rmat=rmatrix)
```

Of course 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`. You need the $B if you don't want these defaults to apply on the next call, otherwise just leave it off.

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, you can use the $O feature.

2. 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 they would like the error messages and other messages to go to separate files.

For the person writing the code which is outputting 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. 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. 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.

3. WITH MULTIPLE LEVELS

With Fortran 2003, one has a very nice way to set up calls to a subroutine package. If one uses a derived type to pass all of the internal information that must be saved from one call to the next, then the code can be made thread safe. If as part of the data structure used, one includes a variable of type `messy_ty`, then at the time just after that structure is initialized the user is free to change defaults as they wish. But the benefits from this sort of organization go deeper.

Suppose a boundary value solver, `bsolve`, calls a nonlinear solver, `nlsolve`, which calls a linear optimizer, `losolve`. Suppose these use corresponding types, `bsolve_ty`, `nlsolve_ty`, `losolve_ty`. If they are all using messy, then each of these types would include a component of type `messy_ty`. `Bsolve` would include a component of type nlsolve_ty and nlsolve would include a component of type `losolve_ty`. Thus when a user declares a type of `bsolve_ty`, all of these other types are included as part of the package. This makes it possible for a routine at the lowest level to get information back to any higher level in a straightforward way. We believe this is an attractive way to design the interface to software packages.

The facility for writing messages to an arbitrary 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.

4. THREADS

If you want to use threads, you should read the comments at the start of the test program `thrdtmessy.f90` to learn what modifications may be needed to `messy_m.f90` in order to use threads. This program gives a very simple example of how such code can be organized if your compiler supports OpenMP. In this code, each thread calls `messy` with a separate variable of derived type `messy_ty`. In the comments there you will find five modes for scheduling threads: STATIC, DYNAMIC, GUIDED, RUNTIME, or AUTO. There is no guarantee that the output files run in different environments will give the same outputs, but all output files as a group should contain all that is written.

A programmer can declare this type to be `threadprivate` or they can allocate an array that corresponds to the maximum number of threads used, as in the example.

5. ACKNOWLEDGMENTS

The author is 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. Richard Hanson has provided useful input, including improvements in exposition, in features and in a careful review of this document. He has also done all of the work for the thread implementation. Tim Hopkins has helped by locating problems in using the NAG compiler.

REFERENCES

Fred T. Krogh. Algorithm xxx: A Fortran Message Processor. *ACM Trans. Math. Softw.* (????).