Fortran_Virtual_Memory

1 SUMMARY

This package **provides read/write facilities for one or more direct-access files through a single in-core buffer,** so that actual input-output operations are often avoided. The buffer is divided into fixed-length *pages* and all input-output is performed by transferring a single page to or from a single record of a file (the length of a record is equal to the length of a page).

Each set of data is addressed as a virtual array, that is, as if it were a very large array. The lower bound of the virtual array is 1. Each element of the virtual array has initial value zero. Any contiguous section of the virtual array may be read or written, without regard to page boundaries.

The virtual array is permitted to be too large to be accommodated in a single file, in which case the package opens secondary files with names that it constructs from the name of the primary file by appending '1', '2', ... We refer to the set of files as a *superfile.* Each superfile is identified by the name of its primary file or the index that it is given when it is opened. To allow the secondary files to reside on different devices, the user is required to supply an array of path names; the full name of a file is the concatenation of a path name with the file name.

To facilitate finite-element assembly and the multifrontal method, there is an option to add data from the virtual array to a given array under the control of a map.

ATTRIBUTES
Authors: J.K. Reid and J.A. Scott, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire OX10 0QX, U.K.
Types: Real (single, double), Complex (single, double), Integer.
Language: Fortran 95 + TR15581 (allocatable components).
Calls: `_COPY`. For efficiency, the user should use an optimised version of this BLAS routine (not required by the integer version).

2 HOW TO USE THE PACKAGE

2.1 The calling sequence

Access to the package requires a `use` statement such as

Single precision version
```
use Fortran_Virtual_Memory_single
```
Double precision version
```
use Fortran_Virtual_Memory_double
```
Integer version
```
use Fortran_Virtual_Memory_integer
```
Complex version
```
use Fortran_Virtual_Memory_complex
```
Double complex version
```
use Fortran_Virtual_Memory_double_complex
```

If it is required to use two modules at the same time, the derived type `FVM_data` (Section 2.1) must be renamed in one of the `use` statements.

There are six entries:

(1) `FVM_initialize` must be called once to initialize a structure of derived type `FVM_data` and to allocate and

initialize its array components.

(2) FVM_open must be called for each superfile that is to be accessed through the package. It gives the superfile an index and opens its files.

(3) FVM_read performs the reading from a superfile.

(4) FVM_write performs the writing to a superfile.

(5) FVM_close should be called for each superfile that is no longer required to be accessed through the package.

(6) FVM_end should be called to deallocate array components of the structure once no further superfiles are required to be accessed through the package.

In the following, INTEGER(long) denotes INTEGER(kind = selected_int_kind(18)) and the **package type** denotes

Default REAL in Fortran_Virtual_Memory_single,

DOUBLE PRECISION in Fortran_Virtual_Memory_double,

Default INTEGER in Fortran_Virtual_Memory_integer,

Default COMPLEX in Fortran_Virtual_Memory_complex.

COMPLEX(kind = kind(0.0d0)) in Fortran_Virtual_Memory_complex.

2.2 The derived type

A single derived type, FVM_data, is accessible from the package. For each instance, the user must declare an object of this type to hold the buffer and all associated data. The object has a large number of components, only some of which can be accessed by the user. These components, which can be accessed by the user but must **not** be altered, are:

entry is a scalar component of type default INTEGER that indicates which of the six entries of the list in Section 2.1 was last invoked.

file_size is a scalar component of type INTEGER(long) that holds the actual size of each file, see argument file_size in Section 2.3.1.

iostat is a scalar component of type default INTEGER that, in the event of an error, holds the Fortran IOSTAT parameter.

lpage is a scalar component of type default INTEGER that holds the size of each page of the in-core buffer, see argument lpage in Section 2.3.1.

ncall_read is a scalar component of type INTEGER(long) that holds the total number of calls made to FVM_read since the call to FVM_initialize.

ncall_write is a scalar component of type INTEGER(long) that holds the total number of calls made to FVM_write since the call to FVM_initialize.

nio_read is a scalar component of type INTEGER(long) that holds the number of records read from files during calls to FVM_read and FVM_write since the call to FVM_initialize.

nio_write is a scalar component of type INTEGER(long) that holds the number of records written to files during calls to FVM_read and FVM_write since the call to FVM_initialize.

npage is a scalar component of type default INTEGER that holds the number of pages in the in-core buffer, see argument lpage in Section 2.3.1.

nwd_read is a scalar component of type INTEGER(long) that holds the the number of scalars of the package type accessed by calls to FVM_read since the call to FVM_initialize.

nwd_write is a scalar component of type INTEGER(long) that holds the the number of scalars of the package type

stored by calls to FVM_write since the call to FVM_initialize.

stat is a scalar component of type default INTEGER that, in the event of an error, holds the Fortran STAT parameter.

2.3 Argument lists

We use square brackets [] to indicate optional arguments. Optional arguments follow the argument data. We recommend that all optional arguments be called by keyword, not by position.

2.3.1 Initialization

This call must be made before any other calls that have the same structure data as an argument.

```
CALL FVM_initialize(iflag,data[,path,file_size,lpage,npage,lp])
```

iflag is a scalar of INTENT(OUT) and type default INTEGER. A successful return is indicated by iflag having the value zero. A negative value is associated with an error message which will be output on unit lp. Possible negative value are:

- -1 Allocation error. The STAT parameter is returned in data%stat.
- -2 One or more of the restrictions file_size≥lpage, lpage≥1, and npage≥1 violated.
- -8 Deallocation error. The STAT parameter is returned in data%stat.
- -16 The character length of path is too great.

data is a scalar of INTENT(OUT) and type FVM_data. On exit, its components will have been initialized. If values other than the default values for npage, lpage, and file_size specified in Section 2.2 are required, the user should use the optional arguments to overwrite the defaults. It must not be altered by the user.

path is an optional assumed-shape rank-one array of INTENT(IN), type default CHARACTER and character length at most 400. If path is absent, the behaviour is as if it were present with the value (/''/). For each superfile named filename that is opened by FVM_open, the name of the primary file is path(i)//filename for an element i of path. Secondary files have names that are constructed by appending 1, 2, ... to this form, perhaps with a different element of path. The value '' is permitted for an element of path. For an old file, the paths are all searched until the file is found. If size(path)>1, there is a check for each new file to make sure that there is room for it, which may be expensive (see Section 4). **Restriction:** len(path)≤400.

file_size is an optional scalar of INTENT(IN) and type INTEGER(long). If present, file_size must be set to the target size each file, measured in scalars of the package type. If absent, the value 2^{21} is used. The actual size is lpage*(file_size/lpage). N.B. This does not limit the size of a superfile which may consist of many files. **Restriction:** file_size≥lpage.

lpage is an optional scalar of INTENT(IN) and type default INTEGER. If present, lpage must be set to the size of each page of the in-core buffer, that is, the number of scalars of the package type in each page and each record of the superfile. If absent, the value $2^{12} = 4096$ is used. **Restriction:** lpage≥1.

npage is an optional scalar of INTENT(IN) and type default INTEGER. If present, npage must hold the number of pages in the in-core buffer associated with data. If absent, the value 1600 is used. **Restriction:** npage≥1.

lp is an optional scalar of INTENT(IN) and type default INTEGER. If present, lp must hold the unit number for diagnostic messages. If not present, lp = 6 is used. If lp is negative, messages are suppressed.

2.3.2 Opening a superfile

This call must be made for each superfile that is to be accessed through the package before any such access.

```
CALL FVM_open(filename,ifile,iflag,data[,lenw,lp])
```

filename is a scalar of INTENT(IN) and type default CHARACTER and length at most 400. On entry, filename must contain the name of the superfile. See argument path in Section 2.3.1 for a description of the naming

convention for the files of a superfile. **Restriction:** len(filename)≤400.

ifile is a scalar of INTENT(OUT) and type default INTEGER. On successful exit, ifile holds the index that it has given to the superfile. The value is positive.

iflag is a scalar of INTENT(OUT) and type default INTEGER. A successful return is indicated by iflag having the value zero. A negative value is associated with an error message which will be output on unit lp. Possible negative values are:

- -1 Allocation error. The STAT parameter is returned in data%stat.
- –5 Error in Fortran INQUIRE statement. The IOSTAT parameter is returned in data%iostat.
- –7 Error in Fortran OPEN statement. The IOSTAT parameter is returned in data%iostat.
- -8 Deallocation error. The STAT parameter is returned in data%stat.
- –11 lenw>0 but either the primary file or one or more of the secondary files does not exist.
- –12 A file of the given name already exists but lenw is not present or lenw≤0. The file path is path(data%iostat).
- –13 The character length of filename is too great.
- –17 The Fortran OPEN statement was not successful for any of the elements of path.

data is a scalar of INTENT(INOUT) and type FVM_data. It must not be altered by the user.

lenw is an optional scalar of INTENT(IN) and type INTEGER(long). If present with a non-positive value or absent, a file of name path(i)//filename must not exist and a new file is created. If present with a positive value, a file of name path(i)//filename must already exist and lenw must hold the length in pages of the part of the virtual array that has been written (that is, the number of records that have been written) and is not to be regarded as having been overwritten by zeros. FVM will behave as if pages beyond this contain zeros. If lenw is so large that more than one file is needed, the appropriate number of secondary files must exist.

lp is an optional scalar of INTENT(IN) and type default INTEGER. If present, lp must hold the unit number for diagnostic messages. If not present, or if lp is equal to the unit number of a file that has already been opened for data, lp = 6 is used. If lp is negative, messages are suppressed.

2.3.3 To read from or write to a virtual array

This call provides read access to a virtual array

```
CALL FVM_read(ifile,loc,n,read_array,iflag,data[,lp,map,discard])
```

and this call provides write access to a virtual array

```
CALL FVM_write(ifile,loc,n,write_array,iflag,data[,lp,inactive])
```

ifile is a scalar of INTENT(IN) and type default INTEGER that holds the index of the superfile, as returned by FVM_open. **Restriction:** ifile>0.

loc is a scalar of INTENT(IN) and type INTEGER(long) that holds the address of the first entry within the virtual array of the data to be read or written. **Restriction:** loc>0.

n is a scalar of INTENT(IN) and type default INTEGER that holds the number of scalars of the package type to be transferred. If n≤0, no action is taken.

read_array is an array of INTENT(INOUT) and of the package type. If map is absent, read_array has size n and superfile data is read into it as if the virtual array were the array virtual_array and the statement

```
read_array(1:n) = virtual_array(loc:loc+n-1)
```

were executed. If map is present, read_array has size at least maxval(map) and superfile data is added into it as if the statement

```
read_array(map(1:n)) = read_array(map(1:n)) + virtual_array(loc:loc+n-1)
```

were executed.

`write_array` is an array of `INTENT(IN)` of size `n` and of the package type. Data is written from `write_array` as if the virtual array were the array `virtual_array` and the statement

```
virtual_array(loc:loc+n-1) = write_array(1:n)
```

were executed.

`iflag` is a scalar of `INTENT(OUT)` and type default `INTEGER`. A successful return is indicated by `iflag` having the value zero. A negative value is associated with an error message which will be output on unit `lp`. Possible negative values are:

- `-3` `loc` is not positive.
- `-4` Attempt to access a superfile that is not open under `Fortran_Virtual_Memory`.
- `-5` Error in Fortran `INQUIRE` statement. The `IOSTAT` parameter is returned in `data%iostat`.
- `-6` Error in Fortran `READ`. The `IOSTAT` parameter is returned in `data%iostat`.
- `-7` Error in Fortran `OPEN` statement. The `IOSTAT` parameter is returned in `data%iostat`.
- `-9` `ifile` less than `1`.
- `-15` Error in Fortran `WRITE`. The `IOSTAT` parameter is returned in `data%iostat`.
- –17 The Fortran `OPEN` statement was not successful for any of the elements of `path`.

`data` is a scalar of `INTENT(INOUT)` and type `FVM_data`. It must not be altered by the user.

`lp` is an optional scalar of `INTENT(IN)` and type default `INTEGER`. If present, `lp` must hold the unit number for diagnostic messages. If not present, or if `lp` is equal to the unit number of a file that has already been opened for `data`, `lp` = `6` is used. If `lp` is negative, messages are suppressed.

`map` is an optional array of `INTENT(IN)`, size `n`, and type default `INTEGER`. Its purpose is explained under the description of `read_array`.

`discard` is an optional scalar of `INTENT(IN)` and type default `LOGICAL`. If present with the value `.TRUE.`, it is assumed that the data will not be read again.

`inactive` is an optional scalar of `INTENT(IN)` and type `INTEGER(long)`. If present, it identifies a range of entries in the superfile that are unlikely to be needed before other data in the buffer. If `inactive` < `loc`, the range is `inactive:loc+n-1`; otherwise, the range is `loc:max(loc+n-1,inactive)`.

2.3.4 Closing a superfile

This call declares that no further access is to be made through the package to a particular superfile. Any information that belongs to the superfile but is in the buffer is optionally transferred to the files and the files are closed. One line of output documenting the action is optionally produced on unit `lp`.

```
CALL FVM_close(ifile,lenw,num_file,iflag,data[,lp,lkeep])
```

`ifile` is a scalar of `INTENT(IN)` and type default `INTEGER` that holds the index the superfile, as returned by `FVM_open`. **Restriction:** `ifile>0`.

`lenw` is a scalar of `INTENT(OUT)` and type `INTEGER(long)`. On exit, `lenw` holds the length in pages of the part of the virtual array that has been written.

`num_file` is a scalar of `INTENT(OUT)` and type default `INTEGER`. On exit, `num_file` is the number of secondary files that have been used.

`iflag` is a scalar of `INTENT(OUT)` and type default `INTEGER`. A successful return is indicated by `iflag` having the value zero. A negative value is associated with an error message which will be output on unit `lp`. Possible

negative values are:

- `-4` Attempt to access a superfile that is not open under `Fortran_Virtual_Memory`.
- `-5` Error in Fortran `INQUIRE` statement. The `IOSTAT` parameter is returned in `data%iostat`.
- `-9` `ifile` less than `1`.
- `-14` Error in Fortran `CLOSE` statement. The `IOSTAT` parameter is returned in `data%iostat`.
- `-15` Error in Fortran `WRITE`. The `IOSTAT` parameter is returned in `data%iostat`.
- –17 The Fortran `OPEN` statement was not successful for any of the elements of `path`.

`data` is a scalar of `INTENT(INOUT)` and type `FVM_data`. It must not be altered by the user.

`lp` is an optional scalar of `INTENT(IN)` and type default `INTEGER`. If present, `lp` must hold the unit number for diagnostic messages. If not present, or if `lp` is equal to the unit number of a file that has already been opened for `data`, `lp` = `6` is used. If `lp` is negative, messages are suppressed.

`lkeep` is an optional scalar of `INTENT(IN)` and type default `LOGICAL`. If present and set to `.FALSE.` or if the superfile has size zero, the files are deleted on being closed. Otherwise, the files are kept.

2.3.5 Final call

This call deallocates the (private) array components of `data`. The call should be made after `FVM_close` has been called for each virtual array accessed through the package with the same structure `data`.

```
CALL FVM_end(iflag,data[,lp])
```

`iflag` is a scalar of `INTENT(OUT)` and type default `INTEGER`. A successful return is indicated by `iflag` having the value zero. A negative value is associated with an error message which will be output on unit `lp`. Possible negative values are:

- `-8` Deallocation error. The `STAT` parameter is returned in `data%stat`.
- `-10` `FVM_close` has not been called for one or more of the superfiles that were opened.

`data` is a scalar of `INTENT(INOUT)` and type `FVM_data`. It must not be altered by the user.

`lp` is an optional scalar of `INTENT(IN)` and type default `INTEGER`. If present, `lp` must hold the unit number for diagnostic messages. If not present, `lp` = `6` is used. If `lp` is negative, messages are suppressed.

3 METHOD

We will refer to a call of `FVM_write` as an ‘active write’ if the argument `inactive` is absent and as an ‘inactive write’ if it is present. We will refer to a call of `FVM_read` as a ‘discarding read’ if the argument `discard` is present with the value `.true.` and as a ‘retaining read’ if `discard` is absent or present with the value `.false.`

The most active pages (records) of the superfile are held in the buffer. We define the *activity* of a page to be our estimate how recently any of its data were written by a active write or read by a retaining read. For each buffer page, the index of the superfile and the page number within the virtual array are stored. Wanted pages are found quickly with the help of a simple hashing function, and hash clashes are resolved by holding doubly-linked lists of pages having identical hash codes. Once the buffer is full and another page is wanted, the least active buffer page is freed. It is identified quickly with the aid of a doubly-linked list of pages in order of last activity (whenever a page is active, it is removed from its old position and placed at the front). A flag is kept for each page to indicate whether it has changed since its entry into the buffer so that only pages which have been changed need be written to superfile when they are freed. On each call of `FVM_read` or `FVM_write`, all wanted pages that are in the buffer are accessed before those that are not in order to avoid freeing a page that may be needed.

For each superfile, a range of discarded entries is kept. If a discarding read touches the discarded range at either

end, the discarded range is expanded to include the newly discarded entries; otherwise, the discarded range consists of the newly discarded entries. If the discarded range is overlapped on an `FVM_write`, it is reset to be null (it was not felt worthwhile to identify a part of the old discarded range that is not overlapped). If a page that is held only in the buffer is found to lie in the discarded range, the page is freed without writing its data to the actual file.

On an inactive write, any page involved that lies entirely within the inactive range is regarded as the least active of the buffer pages; any other page involved is regarded as having unchanged activity.

Note that a call of `FVM_read` may cause an actual write to occur in order to free a page and that a call of `FVM_write` may cause an actual read to occur if only part of the page is changed.

The efficiency is application dependent. If this is important, the user may try several values of `npage` and `lpage`, monitoring the number of actual input/output operations recorded in `data%nio_read` and `data%nio_write`.

The array `path` allows the user to specify that the files may reside on different devices if the superfile is too large for one. When a new file is opened with `size(path)`>1, all the alternatives in `path` are tried until one is found which may be opened, fully written with data, closed, and reopened. If this fails, the next path is tried. Writing the whole file is expensive if it is large, but avoids later failures on writing to the file or closing it; there would be no way to recover from such a failure. If the user is sure that there is enough space, the check is avoided by specifying only one path. Each of the superfiles may still be placed on different devices by suitable choices of `filename` in `FVM_open`.

4 EXAMPLES OF USE

4.1 Simple example

The following simple example opens a superfile, puts 1,2,3,...,20 into positions 10,11,.. , gets data from positions 1,2,...,40 and then closes it. Note that a virtual array is always regarded as starting with all its entries zero, so those entries that are read without being written have the value zero. The code is as follows:

```
program example
      use Fortran_Virtual_Memory_double
      implicit none
      type (FVM_data) :: data
      integer, parameter :: long = selected_int_kind(18)
      integer :: i,ifile,iflag,n,num_file
      integer(long):: lenw,loc
      character(len=8) :: path(1), filename = 'testfile'
      double precision :: array(40)

! Initialize FVM, with path in the current directory
      path(1) = ''
      call FVM_initialize(iflag,data,path)
      if (iflag < 0) stop

! Open superfile
      call FVM_open(filename,ifile,iflag,data)
      if (iflag < 0) go to 20
      n = 20
      do i = 1,n
        array(i) = real(i)
      end do

! Write 1,2,3,...,20 to positions 10,11,... in the superfile
      loc = 10
      call FVM_write(ifile,loc,n,array,iflag,data)
      if (iflag < 0) go to 10

! Read first 40 integers from the superfile and then print them
      loc = 1
      n = 40
```

```
      call FVM_read(ifile,loc,n,array,iflag,data)
      if (iflag < 0) go to 10

      write (6,'(a)') ' Contents of array are:'
      write (6,'(10F8.1)')  array(1:40)

! FVM has finished with superfile
10    call FVM_close(ifile,lenw,num_file,iflag,data,lkeep=.false.)

! Deallocate arrays
20    call FVM_end(iflag,data)

end program example
```

The output produced is

```
 Contents of array are:
     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     1.0
     2.0     3.0     4.0     5.0     6.0     7.0     8.0     9.0    10.0    11.0
    12.0    13.0    14.0    15.0    16.0    17.0    18.0    19.0    20.0     0.0
     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
```

4.2 Example using files of different types

The following example uses two versions of the package, one for double precision reals and one for integers, to illustrate the generic properties. On the `use` statement for `Fortran_Virtual_Memory_integer`, we rename the data type `FWM_data`. This is the only renaming that is needed.

In all the calls of package here, we have added the optional argument `lp` to provoke printing in the event of an error.

For the integer data, we perform the same test as we did for the real data in the simple example. For the real data, we perform a slightly more complicated test involving setting the array to have all values 10.0 and adding values from the file under the control of a map array.

The code is as follows:

```
program example
      use Fortran_Virtual_Memory_double
! Use the integer module with renaming
      use Fortran_Virtual_Memory_integer, FVM_integer_data => FVM_data
      implicit none
      type (FVM_data) :: data
      type (FVM_integer_data) :: intdata
      integer, parameter :: long = selected_int_kind(18)
      integer :: i,ifile(2),iflag,n,num_file
      integer(long):: lenw,loc
      character(len=8) :: path(1), filename(2) = (/ 'realfile', 'intfile '/)
      double precision :: array(40)
      integer :: intarray(40), map(5)

! Initialize FVM, with path in the current directory
      path(1) = ''
      call FVM_initialize(iflag,data,path,lp=6)
      if (iflag < 0) stop
      call FVM_initialize(iflag,intdata,path,lp=6)
      if (iflag < 0) go to 40

! Open superfile
      call FVM_open(filename(1),ifile(1),iflag,data,lp=6)
      if (iflag < 0) go to 30
      call FVM_open(filename(2),ifile(2),iflag,intdata,lp=6)
```

```
      if (iflag < 0) go to 20
      n = 20
      do i = 1,n
        array(i) = real(i)/10
        intarray(i) = i
      end do

! Write to positions 10,11,... in the integer superfile
      loc = 10
      call FVM_write(ifile(2),loc,n,intarray,iflag,intdata,lp=6)
      if (iflag < 0) go to 10

! Write to positions 12,13, ... in the real superfile
      loc = 12
      call FVM_write(ifile(1),loc,n,array,iflag,data,lp=6)
      if (iflag < 0) go to 10

! Set array to 10.0, then add five file values under control of map
      array(:) = 10.0
      n = 5
      map = (/ 5,7,10,22,36 /)
      call FVM_read(ifile(1),loc,n,array,iflag,data,map=map,lp=6)
      if (iflag < 0) go to 10

! Read first 40 values from the integer superfile and print them
      loc = 1
      n = 40
      call FVM_read(ifile(2),loc,n,intarray,iflag,intdata,lp=6)
      if (iflag < 0) go to 10

      write (6,'(a)') ' Contents of arrays are:'
      write (6,'(/,(10F8.1))')  array(1:40)
      write (6,'(/,(10I8))')  intarray(1:40)

! FVM has finished with superfiles
10    call FVM_close(ifile(2),lenw,num_file,iflag,intdata,lkeep=.false.,lp=6)
20    call FVM_close(ifile(1),lenw,num_file,iflag,data,lkeep=.false.,lp=6)

! Deallocate arrays
30    call FVM_end(iflag,intdata,lp=6)
40    call FVM_end(iflag,data,lp=6)

end program example
```

The output produced is

```
 Contents of arrays are:

    10.0    10.0    10.0    10.0    10.1    10.0    10.2    10.0    10.0    10.3
    10.0    10.0    10.0    10.0    10.0    10.0    10.0    10.0    10.0    10.0
    10.0    10.4    10.0    10.0    10.0    10.0    10.0    10.0    10.0    10.0
    10.0    10.0    10.0    10.0    10.0    10.5    10.0    10.0    10.0    10.0

       0       0       0       0       0       0       0       0       0       1
       2       3       4       5       6       7       8       9      10      11
      12      13      14      15      16      17      18      19      20       0
       0       0       0       0       0       0       0       0       0       0
```