EMsoft HDF Routines

March 14, 2017*

Table of Contents

*This document describes all the HDF routines available in the HDFsupport.f90 module. It is an updated version that includes the changes from function to subroutine format for several of the read operations.

1 Introductory comments

Starting with Release 3.0.0, the EMsoft package includes support for the Hierarchical Data Format (HDF5). This support consists mostly of a single module, HDFsupport.f90 in the src folder, which defines all the routines for reading and writing datasets, as well as creating groups and so on. While the H5LT API provides many of the same routines, it was deemed necessary to create a separate module to obtain complete control over all options. This also forced us to enable support for the Fortran-2003 version of the language, which is fully supported by the open source gfortran compiler. The 2003 version provides extensive support for C language communications, as well as access to variables and such across languages. The HDF5 package (currently hdf5-1.8.14 as off March 14, 2017) must be compiled with fortran2003 support enabled, otherwise the HDFsupport module will not compile. In the remainder of this document, we decribe all available functions and subroutines, and provide a few brief examples of writing to and reading from EMsoft HDF files.

2 The HDFsupport.f90 Module

This section enumerates all the available routines in the HDFsupport module. To make use of the module, the following commands must be present at the start of the routine that will make use of them:

```
use typedefs
use HDF5
use HDFsupport
use ISO_C_BINDINGS
```

The latter use-statement enables access to C-type variables and types.

2.1 Push-pop stack

HDF routines typically have a lot of open and close statements for all the types of objects available (file, group, dataset, attribute, etc.). To make things easier for the user, all handling of the associated ID variables is kept hidden from the user through the use of a push-pop stack. The stack is a linked list and items are added to the top of the list in a last-in–first-out approach. Each stack entry is defined by the type:

```
! type definition for HDF-based output
type HDFobjectStackType   ! this is a push-pop stack to keep track of the open objects
  character(LEN=1) :: objectType
  character(fnlen) :: objectName
  integer(HID_T)  :: objectID
  type(HDFobjectStackType),pointer :: next
end type HDFobjectStackType
```

If a routine wants to make use of the HDFsupport module, it must first declare and nullify a variable of this type:

```
  type(HDFobjectStackType),pointer  :: HDF_head

  nullify(HDF_head)
```

Failure to nullify( ) the pointers may result in unpredictable program behavior. The two pointers are arguments to every HDFsupport call and should never be modified by the user program.

The HDF stack must be used by the calling program to close HDF objects; when an object is opened by an HDFsupport routine, its information will be automatically pushed onto the stack, and the calling program does not need to access the stack. Closing an object, however, requires user intervention by means of the HDF_pop routine, which can be called in two ways:

```
call HDF_pop(HDF_head)
```

will close the single object that is currently on the top of the stack. If multiple objects need to be closed, then this call must be made multiple times. It is the responsibility of the programmer to keep track of the hierarchical structure of the HDF file.

When the final write or read has been done, all currently open objects, including the HDF file itself, can be closed by calling:

```
call HDF_pop(HDF_head,.TRUE.)
```

There is hence no need for the programmer to explicitly close an HDF file; a simple HDF_pop call will suffice.

2.2 Initializing the fortran interface

The HDF5 library makes use of many "behind-the-scenes" variables that must be initialized before any HDF operation can be performed. This is done by the following code:

```
!
! Initialize the FORTRAN interface.
!
CALL h5open_EMsoft(hdferr)
if (hdferr.ne.0) then
    ! do some error handling
end if
```

where hdferr is an integer variable that will equal 0 when the initialization was successfull.

At the end of the program, or when no further HDF operations are needed, the fortran interface must be closed to correctly release any HDF-allocated memory resources:

```
!
! Close the FORTRAN interface.
!
CALL h5close_EMsoft(hdferr)
if (hdferr.ne.0) then
    ! do some error handling
end if
```

It is crucially important that the EMsoft programs only open the HDF5 interface once at the start of the program and close it at the end. None of the library routines will perform this action, so it must always be performed in the main calling program. Failure to do so may result in unpredictable program behavior with lots of HDF5 warnings and errors.

In the remainder of this document, we will suppress any further statements of error handling.

2.3 File level operations

The following two function calls can be used to create a new file, and to open an existing file in readonly mode (optional):

```
hdferr = HDF_createFile(HDFname, HDF_head)
hdferr = HDF_openFile(HDFname, HDF_head, readonly)
```

where HDFname is of the type character(fnlen) and is the filename, either the file itself or the full path. Note that the fnlen parameter is defined in the local.f90.in file in the src folder and is currently set to 132 characters, reflecting the limit of characters per line of fortran-90 source code.

The first function above creates the new file and opens it, pushing the file object onto the stack. The second function attempts to open an existing file and also pushes the file object on the stack. At that point, the HDF file is opened into the root / level. Note that the last parameter of the HDF_openFile call is optional, and is of the logical type. Its value (.TRUE. or .FALSE.) is not important; when it is present, file access will be in read-only mode, when it is absent, access will be in read-write mode.

2.4 Group level operations

There are two group level operations, one to create a new group, the other to open an existing group. In both cases, the group object ID will be pushed onto the stack. The routines are:

```
hdferr = HDF_createGroup(groupname, HDF_head)
hdferr = HDF_openGroup(groupname, HDF_head)
```

where groupname is of type character(fnlen). If the HDF_createGroup function is used to access an existing group, then this group is simply opened.

2.5 Dataset level operations

2.5.1 Writing to a dataset

There are 21 write routines, grouped by the type of the dataset that is being written. For strings, we have the following general routine (for brevity, note that the ... are shorthand for HDF_head):

```
hdferr = HDF_writeDatasetStringArray(dataname, stringarray, nlines, HDF_head)
hdferr = HDF_writeDatasetCharArray1D(dataname, chararr, dim0, ...)
hdferr = HDF_writeDatasetCharArray2D(dataname, chararr, dim0, dim1,  ...)
hdferr = HDF_writeDatasetCharArray3D(dataname, chararr, dim0, dim1, dim2,  ...)
hdferr = HDF_writeDatasetCharArray4D(dataname, chararr, dim0, dim1, dim2, dim3,  ...)
```

where dataname is the dataset name (used to create the dataset), stringarray is an array of type character(len=fnlen, KIND=c_char) and has nlines elements; note that to write a single string, this string must be copied into a one-element stringarray(1). The other four routines are used to write an array of individual characters (basically unsigned integers of length 1 byte, representing the interval $[0 \dots 255]$) to the file.

The module also provides a routine to write a complete text file to the HDF file:

```
hdferr = HDF_writeDatasetTextFile(dataname, filename, HDF_head)
```

where filename is of type character(fnlen). This will copy the entire text file into a variable length string array dataset, with each string terminated by the C_NULL_CHAR character. Note that currently the maximum string length is fnlen.

There are three write routines for single variables of the integer, float (=real4), and double (=real8) formats:

```
hdferr = HDF_writeDatasetInteger(dataname, intval, HDF_head)
hdferr = HDF_writeDatasetFloat(dataname, fltval, HDF_head)
hdferr = HDF_writeDatasetDouble(dataname, dblval, HDF_head)
```

where the second argument in each call is the corresponding variable.

Finally, there are routines to write arrays of integers, floats, and doubles to the file; the arrays can be 1D, 2D, 3D, or 4D :

```
hdferr = HDF_writeDatasetIntegerArray1D(dataname, intarr, dim0, ...)
hdferr = HDF_writeDatasetIntegerArray2D(dataname, intarr, dim0, dim1,  ...)
hdferr = HDF_writeDatasetIntegerArray3D(dataname, intarr, dim0, dim1, dim2,  ...)
hdferr = HDF_writeDatasetIntegerArray4D(dataname, intarr, dim0, dim1, dim2, dim3,  ...)

hdferr = HDF_writeDatasetFloatArray1D(dataname, fltarr, dim0,  ...)
hdferr = HDF_writeDatasetFloatArray2D(dataname, fltarr, dim0, dim1,  ...)
hdferr = HDF_writeDatasetFloatArray3D(dataname, fltarr, dim0, dim1, dim2,  ...)
hdferr = HDF_writeDatasetFloatArray4D(dataname, fltarr, dim0, dim1, dim2, dim3,  ...)

hdferr = HDF_writeDatasetDoubleArray1D(dataname, dblarr, dim0,  ...)
hdferr = HDF_writeDatasetDoubleArray2D(dataname, dblarr, dim0, dim1,  ...)
hdferr = HDF_writeDatasetDoubleArray3D(dataname, dblarr, dim0, dim1, dim2,  ...)
hdferr = HDF_writeDatasetDoubleArray4D(dataname, dblarr, dim0, dim1, dim2, dim3,  ...)
```

Note that for each of the HDF_writeDataset... routines described above, one can add an optional keyword at the end of the argument list to indicate that an existing dataset should be overwritten rather than a new one created. This can be done by defining the logical overwrite to be .TRUE., and then adding it as the last argument, e.g.:

```
hdferr = HDF_writeDatasetFloatArray3D(dataname, fltarr, dim0, dim1, dim2, HDF_head, overwrite)
```

In such cases, it is the user's responsibility to ensure that the new data set has exactly the same size as the one being overwritten.

2.5.2 Reading from a dataset

Every write routine from the previous section has a corresponding read version. In all cases, the result of calling these subroutines is directly written into an allocatable array of the proper type, except for the single variable reading subroutines. The main program must not allocate these variables, it should only declare them as allocatable, and then the HDF routines will perform the appropriate allocations with the correct dimensions. The available read routines are (in this case, the ellipses represent HDF_head, hdferr):

```
hdferr = HDF_extractDatasetTextfile(dataname, textfile, ...)  ! the only read function call
call HDF_readDatasetStringArray(dataname, nlines, ..., stringarray)
```

```
call HDF_readDatasetCharArray1D(dataname, dims, ..., chararr1D)
call HDF_readDatasetCharArray2D(dataname, dims, ..., chararr2D)
call HDF_readDatasetCharArray3D(dataname, dims, ..., chararr3D)
call HDF_readDatasetCharArray4D(dataname, dims, ..., chararr4D)

call HDF_readDatasetInteger(dataname, ..., intdata)
call HDF_readDatasetFloat(dataname, ..., fltdata)
call HDF_readDatasetDouble(dataname, ..., dbldata)

call HDF_readDatasetIntegerArray1D(dataname, dims, ..., intarr1D)
call HDF_readDatasetIntegerArray2D(dataname, dims, ..., intarr2D)
call HDF_readDatasetIntegerArray3D(dataname, dims, ..., intarr3D)
call HDF_readDatasetIntegerArray4D(dataname, dims, ..., intarr4D)

call HDF_readDatasetFloatArray1D(dataname, dims, ..., fltarr1D)
call HDF_readDatasetFloatArray2D(dataname, dims, ..., fltarr2D)
call HDF_readDatasetFloatArray3D(dataname, dims, ..., fltarr3D)
call HDF_readDatasetFloatArray4D(dataname, dims, ..., fltarr4D)

call HDF_readDatasetDoubleArray1D(dataname, dims, ..., dblarr1D)
call HDF_readDatasetDoubleArray2D(dataname, dims, ..., dblarr2D)
call HDF_readDatasetDoubleArray3D(dataname, dims, ..., dblarr3D)
call HDF_readDatasetDoubleArray4D(dataname, dims, ..., dblarr4D)
```

The first routine specifically extracts an array of strings from the dataset, and stores it as a textfile with the name textfile; no strings are returned to the calling program.

As an example of the use of these routines, consider a dataset named MyArray, which is of type float (=real4) and has dimensions $400 \times 400 \times 5$. This array can then be read from the file into an allocatable but not yet allocated variable named vals as follows:

```
! in the variable declaration section
real(kind=sgl),allocatable :: vals(:,:,:)
integer(kind=irg) :: dims(3)

! after the dataset has been opened
dataname = 'MyArray'
call HDF_readDatasetFloatArray3D(dataname, dims, HDF_head, hdferr, vals)
```

The array vals then contains the values read from the file, and dims will contain the dimensions of the array.

2.5.3 Writing and reading hyperslabs

In many cases, we do not need to write or read an entire large dataset, but we may want to write it in segments. In HDF parlance, these segments are called "hyperslabs". The 12 routines below create a data space of dimensions hdims (which is an HSIZE_T-type array of 2, 3, or 4 entries), and write the hyperslab wdata of dimensions dim0, dim1, ... at an offset of offset (again an HSIZE_T-type array of 2, 3, or 4 entries) into the larger dataset. It is up to the user to make sure that the dimensions are consistent and will fit in the available space. The final argument insert is an optional logical

argument; when present, the hyperslab will be written to an already existing dataset. So, the first time these functions would be called without the insert argument, and for every additional write to the dataset, the keyword must be present.

```
hdferr = HDF_writeHyperslabCharArray2D(dataname, wdata, hdims, offset, dim0, dim1, ..., [insert])
hdferr = HDF_writeHyperslabCharArray3D(dataname, wdata, hdims, offset, dim0, dim1, dim2, ..., [insert])
hdferr = HDF_writeHyperslabCharArray4D(dataname, wdata, hdims, offset, dim0, dim1, dim2, dim3, ..., [insert])
hdferr = HDF_writeHyperslabIntegerArray2D(dataname, wdata, hdims, offset, dim0, dim1, ..., [insert])
hdferr = HDF_writeHyperslabIntegerArray3D(dataname, wdata, hdims, offset, dim0, dim1, dim2, ..., [insert])
hdferr = HDF_writeHyperslabIntegerArray4D(dataname, wdata, hdims, offset, dim0, dim1, dim2, dim3, ..., [insert])
hdferr = HDF_writeHyperslabFloatArray2D(dataname, wdata, hdims, offset, dim0, dim1, ..., [insert])
hdferr = HDF_writeHyperslabFloatArray3D(dataname, wdata, hdims, offset, dim0, dim1, dim2, ..., [insert])
hdferr = HDF_writeHyperslabFloatArray4D(dataname, wdata, hdims, offset, dim0, dim1, dim2, dim3, ..., [insert])
hdferr = HDF_writeHyperslabDoubleArray2D(dataname, wdata, hdims, offset, dim0, dim1, ..., [insert])
hdferr = HDF_writeHyperslabDoubleArray3D(dataname, wdata, hdims, offset, dim0, dim1, dim2, ..., [insert])
hdferr = HDF_writeHyperslabDoubleArray4D(dataname, wdata, hdims, offset, dim0, dim1, dim2, dim3, ..., [insert])
```

The hyperslabs can be read from the file using the following 12 functions, with arguments already explained above; the functions return a complete array, which must have the status allocatable in the calling program, but must not already be allocated.

```
chararr2D = HDF_readHyperslabCharArray2D(dataname, offset, dims, ...)
chararr3D = HDF_readHyperslabCharArray3D(dataname, offset, dims, ...)
chararr4D = HDF_readHyperslabCharArray4D(dataname, offset, dims, ...)
intarr2D = HDF_readHyperslabIntegerArray2D(dataname, offset, dims, ...)
intarr3D = HDF_readHyperslabIntegerArray3D(dataname, offset, dims, ...)
intarr4D = HDF_readHyperslabIntegerArray4D(dataname, offset, dims, ...)
fltarr2D = HDF_readHyperslabFloatArray2D(dataname, offset, dims, ...)
fltarr3D = HDF_readHyperslabFloatArray3D(dataname, offset, dims, ...)
fltarr4D = HDF_readHyperslabFloatArray4D(dataname, offset, dims, ...)
dblarr2D = HDF_readHyperslabDoubleArray2D(dataname, offset, dims, ...)
dblarr3D = HDF_readHyperslabDoubleArray3D(dataname, offset, dims, ...)
dblarr4D = HDF_readHyperslabDoubleArray4D(dataname, offset, dims, ...)
```

2.6 Example source code

The following example contains code that reads a namelist file for a given program (in this case the EMKossel program and creates an HDF file that contains the entire file in the NMLFiles group, as well as a parsed version of the file in the NMLparameters group. Then the program writes a few data items to the file in the EMData group, and closes the file.

```
program hdf_writetest

  use local
  use HDF5
  use typedefs
  use HDFsupport
  use NameListTypedefs
  use NameListHandlers
  use NameListHDFwriters
  use ISO_C_BINDING

  IMPLICIT NONE
```

```
  CHARACTER(fnlen)                  :: filename, groupname, dataset, nmlname, programname

  CHARACTER(len=fnlen, KIND=c_char), DIMENSION(1), TARGET  :: line
  CHARACTER(len=fnlen, KIND=c_char), ALLOCATABLE, TARGET   :: lines(:)

  character(len=1)                  :: chararr(256)
  character(len=1)                  :: chararr2(256,256)

  INTEGER                           :: i, j, length, nlines, hdferr
  integer(kind=irg)                 :: intarr(8), dim0, dim1, intarr2(3,3), dims(1), dims2(2), &
                                       intdata(100,100), sdata(10,10), offset(2), cnt(2)
  real(kind=sgl)                    :: fltarr(8), realdata(20, 20)
  real(kind=dbl)                    :: dblarr(8)

  type(HDFobjectStackType),pointer  :: HDF_head
  type(HDFobjectStackType),pointer  :: HDF_tail
  type(KosselNameListType)          :: knl

  character(11)                     :: dstr
  character(15)                     :: tstrb
  character(15)                     :: tstre

  nullify(HDF_head)
  nullify(HDF_tail)

! read the namelist file into the knl structure
nmlname = 'EMKossel.nml'
call GetKosselNameList(nmlname,knl)

! get date and time stamps
call timestamp(datestring=dstr, timestring=tstrb)

! since there is no actual computation involved, set end time equal to start time
tstre = tstrb

! Initialize FORTRAN interface.
CALL h5open_EMsoft(hdferr)

! Create a new HDF5 file
filename = 'HDFtest.h5'
hdferr =  HDF_createFile(filename, HDF_head)

! write the EMheader to the file
programname = 'hdf_writetest.f90'
call HDF_writeEMheader(HDF_head, dstr, tstrb, tstre, programname)

! create a namelist group to write all the namelist files into
groupname = "NMLfiles"
hdferr = HDF_createGroup(groupname, HDF_head)

! read the text file and write the array to the file
  dataset = 'KosselNML'
  hdferr = HDF_writeDatasetTextFile(dataset, nmlname, HDF_head)
```

```
! leave this group
call HDF_pop(HDF_head)

! create a NMLparameters group to write all the namelist entries into
groupname = "NMLparameters"
hdferr = HDF_createGroup(groupname, HDF_head)

  call HDFwriteKosselNameList(HDF_head, knl)

! and leave this group
call HDF_pop(HDF_head)

! then the remainder of the data in a EMData group
groupname = 'EMData'
hdferr = HDF_createGroup(groupname, HDF_head)

allocate(lines(2))
lines(1) = 'This is line 1'
lines(2) = 'and this is line 2'
dataset = 'StringTest'
hdferr = HDF_writeDatasetStringArray(dataset, lines, 2, HDF_head)
deallocate(lines)

intarr = (/ 1, 2, 3, 4, 5, 6, 7, 8 /)
dataset = 'intarr1D'
dims = shape(intarr)
dim0 = dims(1)
hdferr = HDF_writeDatasetIntegerArray1D(dataset, intarr, dim0, HDF_head)

intarr2 = reshape( (/ 1,2,3,4,5,6,7,8,9 /), (/3,3/))
dataset = 'intarr2D'
dims2 = shape(intarr2)
dim0 = dims2(1)
dim1 = dims2(2)
hdferr = HDF_writeDatasetIntegerArray2D(dataset, intarr2, dim0, dim1, HDF_head)

fltarr = (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 /)
dataset = 'fltarr1D'
dims = shape(fltarr)
dim0 = dims(1)
hdferr = HDF_writeDatasetFloatArray1D(dataset, fltarr, dim0, HDF_head)

dblarr = (/ 1.D0, 2.D0, 3.D0, 4.D0, 5.D0, 6.D0, 7.D0, 8.D0 /)
dataset = 'dblarr1D'
dims = shape(dblarr)
dim0 = dims(1)
hdferr = HDF_writeDatasetDoubleArray1D(dataset, dblarr, dim0, HDF_head)

! some character arrays
do i1=0,255
  chararr(i1+1) = char(i1)
end do
```

```
dataset = 'chararray1D'
dim0 = 256
hdferr = HDF_writeDatasetCharArray1D(dataset, chararr, dim0, HDF_head)

do i1=0,255
 do i2=0,255
  chararr2(i1+1,i2+1) = char(mod(i1+i2,256))
 end do
end do

dataset = 'chararray2D'
dim0 = 256
hdferr = HDF_writeDatasetCharArray2D(dataset, chararr2, dim0, dim0, HDF_head)

do i = 1, 100
  do j = 1, 100
    intdata(i,j) = (i-1) + (j-1);
  end do
end do

! and finally two hyperslab examples
dataset = 'hyperslab'
sdata = intdata(1:10,1:10)
dims2 = (/ 1000, 1000 /)
cnt = (/ 10, 10 /)
offset = (/ 5, 5 /)
hdferr = HDF_writeHyperslabIntegerArray2D(dataset, sdata, dims2, offset, cnt(1), cnt(2), HDF_head)
insert = .TRUE.
offset = (/ 0, 0 /)  ! this will partially overwrite the previous hyperslab
hdferr = HDF_writeHyperslabIntegerArray2D(dataset, sdata, dims2, offset, cnt(1), cnt(2), HDF_head, inse

do i=1,20
  do j=1,20
    realdata(i,j) = float(i-1) + float(j+1)
  end do
end do

dataset = 'realhyperslab'
dims2 = (/ 100, 100 /)
cnt = (/ 20, 20 /)
offset = (/ 2, 2 /)
hdferr = HDF_writeHyperslabFloatArray2D(dataset, realdata, dims2, offset, cnt(1), cnt(2), HDF_head)

! close all objects, including the file
call HDF_pop(HDF_head,.TRUE.)

! close the Fortran interface
call h5close_EMsoft(hdferr)

end program hdf_writetest
```

This creates a file HDFtest.h5 than can be analyzed with the java-based HDFView program, available from the HDF group web site. Fig. 1 shows the internal layout of the file, with all the groups and datasets.

In the second example, this file is opened to read some of the data entries and write them to the console.

```
program hdf_readtest

  use local
  use HDF5
  use typedefs
  use HDFsupport
  use NameListTypedefs
  use NameListHandlers
  use NameListHDFwriters
  use ISO_C_BINDING

  IMPLICIT NONE

  character(fnlen)                :: filename, groupname, dataset, textfile

  character(len=fnlen, KIND=c_char), ALLOCATABLE, TARGET   :: lines(:)

  integer(HSIZE_T)                 :: dims(1), dims2(2)

  integer                          :: i, j, length, nlines, hdferr, offset(2), cnt(2)
  integer(kind=irg),allocatable    :: rdintarr(:), rdintarr2(:,:), intarr10(:,:)
  real(kind=sgl),allocatable       :: rdfltarr(:), realarr10(:,:)
  real(kind=dbl),allocatable       :: rddblarr(:)
  character(len=1),allocatable     :: rdchararr(:)

  type(HDFobjectStackType),pointer  :: HDF_head
  type(HDFobjectStackType),pointer  :: HDF_tail
  type(KosselNameListType)          :: knl

  character(11)                    :: dstr
  character(15)                    :: tstrb
  character(15)                    :: tstre

  nullify(HDF_head)
  nullify(HDF_tail)

! Initialize FORTRAN interface.
CALL h5open_EMsoft(hdferr)

! Open the file using the default properties.
filename = 'HDFtest.h5'
hdferr =  HDF_openFile(filename, HDF_head)

! open the NMLfiles group
groupname = 'NMLfiles'
hdferr = HDF_OpenGroup(groupname, HDF_head)

! read a dataset
  dataset = 'KosselNML'
  call HDF_readDatasetStringArray(dataset, nlines, HDF_head, hdferr, lines)
  write (*,*) 'data set name : ',trim(dataset),':  number of lines read = ',nlines
```

```
  do i=1,nlines
    write (*,*) lines(i)
  end do
  deallocate(lines)

! extract and write the KosselNML array to a test file
  textfile = 'test.nml'
  hdferr = HDF_extractDatasetTextfile(dataset, textfile, HDF_head)

call HDF_pop(HDF_head)

! next, read one of the integer string arrays

! open the EMData group
groupname = 'EMData'
hdferr = HDF_OpenGroup(groupname, HDF_head)

! remember that the read routines do the allocations !
dataset = 'StringTest'
call HDF_readDatasetStringArray(dataset, nlines, HDF_head, hdferr, lines)
do i=1,nlines
  write (*,*) lines(i)
end do
deallocate(lines)

dataset = 'intarr1D'
call HDF_readDatasetIntegerArray1D(dataset, dims, HDF_head, hdferr, rdintarr)

write (*,*) 'shape of read intarr = ',shape(rdintarr)
write (*,*) rdintarr
deallocate(rdintarr)

dataset = 'intarr2D'
call HDF_readDatasetIntegerArray2D(dataset, dims2, HDF_head, hdferr, rdintarr2)

write (*,*) 'shape of read intarr2 = ',shape(rdintarr2)
write (*,*) rdintarr2
deallocate(rdintarr2)

dataset = 'fltarr1D'
call HDF_readDatasetFloatArray1D(dataset, dims, HDF_head, hdferr, rdfltarr)

write (*,*) 'shape of read fltarr = ',shape(rdfltarr)
write (*,*) rdfltarr
deallocate(rdfltarr)

dataset = 'dblarr1D'
call HDF_readDatasetDoubleArray1D(dataset, dims, HDF_head, hdferr, rddblarr)

write (*,*) 'shape of read dblarr = ',shape(rddblarr)
write (*,*) rddblarr
deallocate(rddblarr)
```

```
dataname = 'chararray1D'
call HDF_readDatasetCharArray1D(dataname, dims, HDF_head, dims, rdchararr)

write (*,*) 'shape of chararray = ',shape(rdchararr)
write (*,*) rdchararr

! and finally, read a hyperslab from the hyperslab dataset
dataname = 'hyperslab'
offset = (/ 3, 3 /)
cnt = (/ 10, 10 /)
intarr10 = HDF_readHyperslabIntegerArray2D(dataname, offset, cnt, HDF_head)

write (*,*) 'hyperslab read : '
do i=1,10
  write (*,"(10I3)") (intarr10(i,j),j=1,10)
end do

dataname = 'realhyperslab'
offset = (/ 2, 2 /)
cnt = (/ 20, 20 /)
realarr10 = HDF_readHyperslabFloatArray2D(dataname, offset, cnt, HDF_head)

write (*,*) 'real hyperslab read : '
do i=1,10
  write (*,"(20F6.1)") (realarr10(i,j),j=1,20)
end do

! close all objects, including the file
call HDF_pop(HDF_head,.TRUE.)

! close the Fortran interface
call h5close_EMsoft(hdferr)

end program hdf_readtest
```

Note that the dims and dims2 arrays are of the HDF integer type HSIZE_T, which is an 8-byte integer.

3 Crystal structure data in HDF5 format

In earlier releases, the crystal structure data was stored in binary form in files of the type *.xtal. A series of new routines was created to store crystal structure data in HDF5 format, which makes the files editable by the HDFView program. These files are also stored in a single folder EMdatapathname/XtalFolder; in the old approach, *.xtal files would be scattered all over the filesystem. A mechanism is provided that recognizes data files in the old format, and automatically converts them to the new HDF5 format

New structure files are generated with the EMmkxtal program. In contrast with earlier versions of the programs, this is now the only way to create a structure input file. The output of the EMmkxtal program is an HDF file (suggested extension: .xtal) with the internal structure shown in Fig. 2.

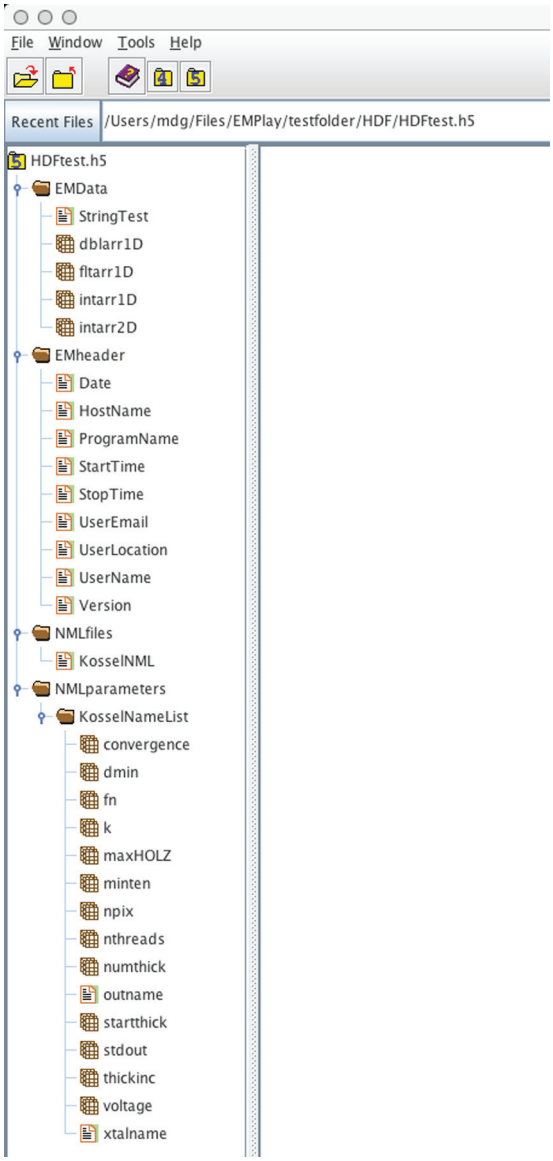

Figure 1: *Internal layout of the HDFtest.h5 file as represented by the Java HDFView program.*

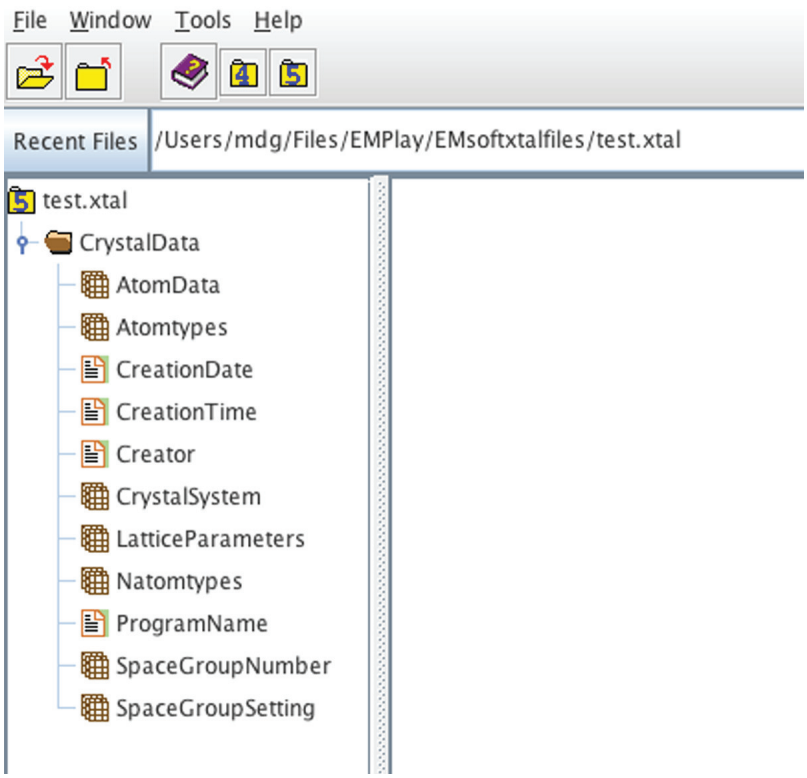

Figure 2: *Internal layout of a typical *.xtal structure file as represented by the Java HDFView program.*