New Features in the HDF5 Fortran Library:
Adding support for the Fortran 2003 Standard

This document describes limitations in the current HDF5 Fortran Library and how they are addressed in HDF5 Release 1.8.8 by using properties of the Fortran 2003 standard. The release is scheduled for November 2011.

The 1.8.8 version of the library supports a wider set of Fortran data types and HDF5 datatypes[1] than prior versions. The wider set includes the following:

- Any kind of Fortran `INTEGER` or `REAL`
- Fortran derived types
- Fortran enumerations
- HDF5 datatypes
    - Enum datatype
    - Variable-length datatype
    - Compound datatype of any complexity

The 1.8.8 version of the library also contains several new Fortran bindings for the HDF5 C functions that use a callback function as a parameter.

[1] We use “data type” when referring to a Fortran language type and “datatype” when referring to an HDF5 type.

Contents

1. Introduction

The HDF5 Fortran APIs were first introduced more than 10 years ago in HDF5 Release 1.4.0. The initial implementation has used Fortran 90 features such as modules, function overloading, function interfaces, dynamic memory allocation, and optional parameters. In many cases, this made the Fortran APIs simpler than their C counterparts. While the choice of the Fortran 90 standard for the implementation of bindings allowed us to provide one source code for all platforms and to take advantage of compiler-level protection, it also restricted us from supporting some HDF5 features available for applications written in C or C++. For example, we could not support HDF5 compound and enumeration datatypes or HDF5 APIs that use a callback function as an argument.

In the past few years most of the commercial and free Fortran compilers have added support for the Fortran 2003 standard. The standard provides interoperability between C and Fortran, and this interoperability makes it possible for us to improve the HDF5 Fortran APIs. The new features described in this document will be available in HDF5 Release 1.8.8. Readers interested in testing and commenting on the features described below may access the source code at https://svn.hdfgroup.uiuc.edu/hdf5/branches/hdf5_1_8 or pick up the latest snapshot from http://www.hdfgroup.uiuc.edu/ftp/pub/outgoing/hdf5/snapshots/v18/. Please send any comments or questions to the HDF Group Help Desk via help@hdfgroup.org.

2. Limitations of the Current Implementation

The HDF5 Fortran bindings in versions prior to 1.8.8 have at least two limitations. First, they support only a small number of the HDF5 datatypes available to C applications, and second, many powerful HDF5 C functions that have callback functions in their parameters do not have Fortran counterparts. For example, `H5Literate` has not had a Fortran counterpart. For more information, see http://www.hdfgroup.org/HDF5/doc/RM/RM_H5L.html#Link-Iterate. The sections below provide more background on these issues.

2.1. Support for Intrinsic Fortran Data Types

HDF5 Fortran applications have only been able to write or read data using the following four intrinsic data types: `INTEGER`, `REAL`, `CHARACTER`, and `DOUBLE PRECISION`. `DOUBLE PRECISION` is obsolete and is not recommended by Fortran 90 and later standards. Applications cannot pass to HDF5 any data buffers that have a "non-standard" type like `INTEGER*1`, `INTEGER*2`, `INTEGER*8`, or `REAL*16`. While those types are non-standard and may be expected to be unsupported, the HDF5 Fortran Library has not supported Fortran 90 recommended integers, reals, and characters of different kinds either. This is a severe limitation, and sometimes there is no workaround. For example, if an HDF5 dataset stores 64-bit integers, data cannot be read by an HDF5 Fortran application into the buffer of the appropriate type like `INTEGER*8` or `INTEGER(KIND=big_enough_to_store)` because there is no interface provided by the HDF5 Fortran Library. Data can be read into the `INTEGER` buffer instead, but precision may be lost. Another example is storing or retrieving data that requires one or two bytes of storage for each data element. The application has to use an `INTEGER` buffer to hold the data and rely on the HDF5 Library to perform the datatype conversion. As a result, the application uses more memory than necessary since `INTEGER` is usually 4 bytes, and the library performs extra work to convert data between the memory buffer and the file.

2.2. Support for Fortran Derived Data Types and HDF5 Compound Datatypes

Fortran 90 introduced derived data types that are similar to C structures. This is shown in the following example:

C Structure	Fortran 90 Derived Data Type

`typedef struct {`
`  int a;`
`  float b;`
`  my_type c;`
`}`
	`TYPE DERIVED`
	`  INTEGER A`
	`  REAL B`
	`  TYPE(MY_TYPE) C`
	`END TYPE DERIVED`

Unlike the intrinsic types that are defined by the language, derived data types are defined by the programmer. While it is usually possible to construct a corresponding HDF5 compound datatype, data of a derived type cannot be passed to current HDF5 subroutines for writing and reading because there is no predefined module procedure due to the arbitrariness of the type. For the same reason, Fortran applications cannot write and

read data using HDF5 compound datatypes if the datatypes contain Fortran data types other than the Fortran intrinsic types supported in the library prior to 1.8.8. Writing and reading data of HDF5 compound datatypes is very inefficient since the application has to write and read data one field at a time field. See the example at http://www.hdfgroup.org/HDF5/Tutor/examples/F90/compound.f90.

2.3. Support for HDF5 Variable-length Datatypes

Versions of the HDF5 Fortran Library prior to 1.8.8 have provided an inefficient way of managing a limited number of HDF5 variable-length datatypes. The base type of the HDF5 variable-length datatype has to be one of the Fortran intrinsic types. Applications have to use the special subroutines `h5dwrite_vl_f` and `h5dread_vl_f` to perform I/O. Both subroutines take as parameters two arrays (a two-dimensional array `buf(max_elem_length, num_elem)` with the actual data and a one-dimensional array with the length of each element `elen(num_elem)`) and pass this data to a wrapper layer that repacks the application's data into an appropriate C structure. This approach is not scalable; it is also not convenient because a special subroutine has to be called to perform I/O on a variable-length data.

2.4. Support for Fortran Enumerated Data Types and HDF5 Enumerated Datatypes

The HDF5 C Library supports enumerated datatypes. See the "Datatypes" chapter in the **HDF5 User's Guide** at http://www.hdfgroup.org/HDF5/doc/UG/. Versions of the HDF5 Fortran Library prior to 1.8.8 cannot write or read data of such type.

Fortran 2003 introduced an enumeration definition to allow interoperability with enumeration constants in C. An example of a Fortran enumerator and its C counterpart are shown below:

C	**Fortran 2003**

`typedef enum {`
`  A = 65;`
`  D = 68;`
`  E;`
`} ascii_code_t`
	`ENUM, BIND(C)`
	`  ENUMERATOR :: A = 65, D = 68`
	`  ENUMERATOR e`
	`END ENUM`

Both definitions declare an enumerator with constants 65, 68, and 69. The Fortran 2003 standard guarantees that constants declared as enumerator correspond to the same integer type used in C; in other words, `int`. This new feature allows us to support HDF5 enumerated datatypes in version 1.8.8.

2.5. HDF5 APIs with Callback Functions

The HDF5 Fortran Library prior to the 1.8.8 release has not had Fortran counterparts of the C functions that use callbacks as parameters. For example, there is no Fortran subroutine for the `H5Literate` function. As a result, operations such as traversing an HDF5 file, customizing datatype conversions, handling error stacks, and controlling metadata caches have not been available to Fortran application developers.

3. Support for Fortran 2003 Features in HDF5

Fortran 2003 provides a standard mechanism for interoperability with C. The limitations discussed in section 2 are addressed by expanding the Fortran API in HDF5-1.8.8 to support C data and function pointers.

An HDF5 Fortran program on its part has to treat all previously unsupported types, variables, and procedures that will be passed to the HDF5 Fortran API as 'interoperable' with C. This is done by following the programming model:

1. Include the `ISO_C_BINDING` module
2. Use the following declarations for variables and functions:
    a. Use the `TARGET` attribute in declarations of a variable or an array that contains data to be written or read by the HDF5 Fortran APIs
    b. Use the `C_PTR` derived data type to declare a pointer to a variable or array in item a
    c. Use the `BIND(C)` attribute in derived type declarations
    d. Use the `BIND(C)` attribute in a Fortran callback function declaration
    e. Use the `C_FUNPTR` derived type to declare a pointer to the Fortran function in item d
3. Associate a pointer with a variable or an array using the `C_LOC` intrinsic data type, and then pass it an HDF5 Fortran call
4. Associate a pointer with a callback function using the `C_FUNLOC` intrinsic data type, and then pass it an HDF5 Fortran call

The example below shows how to pass a buffer.

```
PROGRAM main
  USE ISO_C_BINDING
  USE HDF5
  …
  TYPE, BIND(C) :: sensor_t
  …
  END TYPE sensor_t
  TYPE(sensor_t), DIMENSION(1:100), TARGET ::  wdata ! Write buffer
  TYPE(C_PTR) :: ptr
  …
  ptr = C_LOC(wdata(1))
  CALL h5dwrite_f(dset, memtype, ptr, hdferr)
  …
END PROGRAM main
```

The example below shows how to pass a callback function.

```
PROGRAM main
  USE ISO_C_BINDING
  USE HDF5
! Type iter_info and call op_func function are declared in liter_cb_mod
  USE liter_cb_mod
  …
  TYPE(C_PTR) :: ptr
  TYPE(C_FUNPTR) :: funptr
  TYPE(iter_info), TARGET :: info
  …
```

```
      ptr = C_LOC(info)
      funptr = C_FUNLOC(op_func)
      CALL h5literate_f(file, …, funptr, ptr, …)
      …
    END PROGRAM main
```

To enable Fortran 2003 features in HDF5, use the `--enable-fortran2003` configure flag in addition to the `--enable-fortran` flag when configuring the HDF5 Library. Configure checks to see if the Fortran compiler is compliant with the Fortran 2003 standard and enables support for new features; if not, configure will fail. You can also check the summary in the `libhdf5.settings` file found in the `lib` subdirectory under the installation point to check for Fortran 2003 support. See the example below.

```
            SUMMARY OF THE HDF5 CONFIGURATION
            =================================
…

Languages:
----------
                        Fortran: yes
                Fortran Compiler: /usr/local/bin/gfortran …
          Fortran 2003 Compiler: yes
 …
```

The current implementation of the Fortran 2003 features was tested with the following compilers:

Operating System	**Fortran Compiler**
Linux 32- and 64-bit systems	gfortran 4.5.* PGI Fortran 11.7 and 11.8 Intel 11.1 and 12.0
Mac OS X	gfortran 4.6.*
SunOS	Oracle Studio 12.3 beta

4. New Capabilities of the HDF5 Fortran Library

This section describes in more detail how different Fortran data types, HDF5 datatypes, and HDF5 callback functions are handled in HDF5 Release 1.8.8. The source code of the examples in this section can be downloaded from http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/api18-fortran.html. Check for the files with names containing the "F03" string.

4.1. Fortran INTEGER and REAL Data Types

Fortran has five intrinsic data types: INTEGER, REAL, COMPLEX, CHARACTER, and LOGICAL. The Fortran 90 standard introduced a property called KIND that characterized precision and range for the first three types and storage presentation for the last two. Only one kind is required by the standard, but a processor may provide more. For example, very often there will be two kinds of REAL data types that correspond to single and double precision.

The HDF5 Fortran Library before version 1.8.8 could handle only INTEGER, REAL, and CHARACTER types, and an obsolete DOUBLE PRECISION type. It could not support COMPLEX and LOGICAL types because there is no support for the corresponding C types in the HDF5 C Library. The library also does not support intrinsic types of a non-default kind. For example, if a processor supports a one-byte integer type and an application has to store integers with the values between -128 to 127, it has to use INTEGER type buffers and rely on HDF5 to perform conversion to store one-byte integers in the HDF5 file.

The C interoperability that was introduced by the Fortran 2003 standard allows the HDF5 Fortran Library to support any kind of INTEGER or REAL types as discussed in the following sections.

4.1.1. Function to Convert an Intrinsic INTEGER or REAL Fortran Data Type to an HDF5 Datatype

The HDF5 Fortran Library provides predefined HDF5 datatypes that correspond to the Fortran intrinsic types INTEGER, REAL, and CHARACTER: H5T_NATIVE_INTEGER, H5T_NATIVE_REAL, and H5T_NATIVE_CHARACTER. There are no HDF5 predefined types for kinds of INTEGERS and REALS that may be available on the system. To find a corresponding HDF5 datatype, use the new function h5kind_to_type. The signature is shown below:

```
INTEGER(HID_T) FUNCTION h5kind_to_type(kind, flag) RESULT(h5_type)
```

The flag parameter can be either H5_INTEGER_KIND or H5_REAL_KIND. For example, suppose an application uses an integer variable array declared as the following:

```
INTEGER(SELECTED_INT_KIND (5)), DIMENSION(100), TARGET :: ivar
```

A Fortran HDF5 application should find the corresponding HDF5 datatype by using the following call:

```
mem_type = h5kind_to_type(KIND(ivar(1), H5_INTEGER_KIND)
```

The returned type could then be used to describe a memory buffer in an HDF5 API call. The example below uses the h5dwrite_f function:

```
ptr = C_LOC(ivar(1))
CALL h5dwrite_f (dset_t, mem_type, ptr, error)
```

The example in the section below illustrates the usage of the function h5kind_to_type.

4.1.2. Example of the h5kind_to_type Function

The example program h5ex_d_rdwr_kind_F03.f90 shows how to read and write real and integer data where the precision is set by SELECTED_REAL_KIND and SELECTED_INT_KIND. The whole example program is at http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/api18-fortran.html.

First, the program defines precision for a REAL type and the range for an INTEGER type, and then it declares array variables as shown below.

```
INTEGER, PARAMETER :: sp = KIND(1.0),   &
dp = SELECTED_REAL_KIND(2*PRECISION(1.0_sp)))
INTEGER, PARAMETER :: ip = SELECTED_INT_KIND(10)

REAL(KIND=dp), DIMENSION(…), TARGET :: wdata_r   ! Double precision floating point
INTEGER(KIND=ip), DIMENSION(…), TARGET :: wdata_i! Integer between -10^10 and 10^10
```

Second, the program gets the corresponding HDF5 datatypes by calling h5kind_to_type, and the result is used in the calls to h5dcreate_f and h5dwrite_f.

```
h5_kind_type_r = h5kind_to_type(dp,H5_REAL_KIND)
h5_kind_type_i = h5kind_to_type(ip,H5_INTEGER_KIND)
```

Then the datasets are created and written.

```
CALL h5dcreate_f(file, dataset_r, h5_kind_type_r, space, dset_r, hdferr)
CALL h5dcreate_f(file, dataset_i, h5_kind_type_i, space, dset_i, hdferr)
….
CALL h5dwrite_f(dset_i, h5_kind_type_i, C_LOC(wdata_i(1,1)), hdferr)
CALL h5dwrite_f(dset_r, h5_kind_type_r, C_LOC(wdata_r(1,1)),  hdferr)
```

4.2. Compound Datatypes

Fortran 90 derived data types are similar to C structures. The example below shows a declaration of a derived type `sensor_t` with `INTEGER`, `CHARACTER`, and `DOUBLE PRECISION` members and shows the initialization of the variable of this type. See `h5ex_t_cmpd_F03.f90` at http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/api18-fortran.html for the complete example.

```
  TYPE! Compound data type
     INTEGER :: serial_no
     CHARACTER(LEN=maxstringlen) :: location
     REAL(real_kind_15) :: temperature
     REAL(real_kind_15) :: pressure
  END TYPE sensor_t

TYPE(sensor_t), DIMENSION(1:dim0), TARGET ::  wdata
  wdata(1)%serial_no = 1153
  wdata(1)%location = "Exterior (static)"
  wdata(1)%temperature = 53.23_real_kind_15
  wdata(1)%pressure = 24.57_real_kind_15
….
```

Prior to HDF5 Release 1.8.8, HDF5 Fortran applications could not easily store derived type data using HDF5 compound datatypes. Derived types had to be composed with the members of the default Fortran intrinsic types and had to be written by a field. Construction of the compound type in Fortran also presented some difficulties since it required manual calculations of the members' offsets. For details, see the example in the HDF5 Tutorial on the compound datatypes at http://www.hdfgroup.org/HDF5/Tutor/compound.html.

The Fortran 2003 standard allows the HDF5 Fortran Library to enable an easy and efficient way to work with compound datatypes. The sections below show how to construct an HDF5 compound datatype that corresponds to the derived type and how to write and read data of this type.

4.2.1. Constructing a Compound Datatype with `H5OFFSETOF`

As mentioned above, versions of the HDF5 Fortran Library prior to 1.8.8 require manual calculations of the members' offsets within a structure when an HDF5 compound datatype is created. The 1.8.8 version provides the `H5OFFSETOF` function to find offsets. This function is similar to the HDF5 C Library macro `HOFFSET`. The following is the signature:

```
FUNCTION h5offsetof(structure_ptr,member_ptr) RESULT(member_offset)
```

`structure_ptr` is a C address of the derived type element, and `member_ptr` is a C address of its member. The size returned by the function is used with the `h5tinsert_f` function to specify the offset of a member within the derived type.

The example below illustrates the calculation of an offset of the `pressure` member and how it is passed it to the `h5tinsert_f` call for constructing an HDF5 compound datatype.

```
…
CALL h5tinsert_f(memtype, "pressure", &
H5OFFSETOF(C_LOC(wdata(1)),C_LOC(wdata(1)%pressure)), H5T_NATIVE_INTEGER, hdferr)
…..
```

4.2.2. How to Write or Read a Compound Datatype

After a memory datatype was constructed as shown in section 4.2.1, data can be written by passing a C pointer to the `h5dwrite_f` call (or `h5dread_f` call).

```
CALL h5dwrite_f(dset, memtype, C_LOC(wdata(1)), hdferr)
```

Please notice a simplified interface: there is no longer any need to pass the `dims` parameter when passing data by C pointer. The same is true for writing and reading HDF5 attributes of compound datatypes; for more details see the example `h5ex_t_cmpd_F03.f90` at http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/api18-fortran.html

4.2.3. Variable-length Datatypes

In this section we will show how to write and read data of the variable-length datatype using `h5dwrite_f` and `h5dread_f` APIs instead on the specialized `h5dwrite_vl_f` and `h5dread_vl_f` APIs. First, we will look at the writing and reading of variable-length strings. Then we will discuss the writing and reading of variable-length data of an arbitrary base datatype.

4.2.3.1. Steps to Write or Read Variable-length Strings

The example `h5ex_t_vlstring.f90` shows how to store an array of Fortran strings as variable-length C strings in an HDF5 file and how to read them back using `h5dwrite_vl_f` and `h5dread_vl_f` APIs. This approach requires usage of special API and advance knowledge of the maximum length of all strings to be written. The example `h5ex_t_vlstring_F03.f90` shows how to write and read Fortran strings that have different lengths without using special APIs.

4.2.3.1.1. Declaring Variable-length Strings

To write Fortran strings of different lengths, one has to declare a write buffer (in our example wdata) and initialize it as follows:

```
TYPE(C_PTR), DIMENSION(1:dim0) :: wdata
CHARACTER(len=8, KIND=c_char), DIMENSION(1), TARGET  :: A = "Parting"//C_NULL_CHAR
…
CHARACTER(len=6, KIND=c_char), DIMENSION(1), TARGET  :: C = "sweet"//C_NULL_CHAR
! Initialize array of C pointers
wdata(1) = C_LOC(A(1))
  …
wdata(3) = C_LOC(C(1))
```

Please notice the usage of the KIND parameter and its value c_char. The length of the original string is increased by 1 (specifying 8 instead of 7), and C_NULL_CHAR is added to the end of the Fortran string. The write buffer is initialized with a C address of the modified Fortran string.

The array for reading data back is declared as follows:

```
TYPE(C_PTR), DIMENSION(:), ALLOCATABLE :: rdata
```

4.2.3.1.2. Writing and Reading Variable-length Strings

After variables were declared and initialized in the section above (4.2.3.1.1), dataset creation to store data and write or read data is straightforward:

```
CALL h5dcreate_f(file, dataset, H5T_STRING, space, dset, hdferr)
CALL h5dwrite_f(dset, H5T_STRING, wdata, hdferr)
…
ALLOCATE(rdata(1:dims(1)))
CALL h5dread_f(dset, H5T_STRING, rdata, hdferr)
```

Since the call above returns a pointer array, we will need to get a Fortran pointer for each element of the array to get a string. This can be done by using the C_F_POINTER intrinsic procedure as shown below:

```
CHARACTER(len=8, kind=c_char), POINTER :: data
….
  DO i = 1, dims(1)
     CALL C_F_POINTER(rdata(i), data)
  ! Display data
  END DO
```

If the length in the declaration of the data pointer is not big enough, the string will be truncated.

4.2.3.2. Steps to Write or Read Variable-length Data

Adding support for the C_PTR derived type made it easy to support HDF5 variable-length datatypes of any base type. The HDF5 Fortran library introduced new derived type hvl_t defined as

```
TYPE hvl_t
     INTEGER(size_t) :: len ! Length of VL data (in base type units)
     TYPE(C_PTR) :: p       ! Pointer to VL data
  END TYPE hvl_t
```

4.2.3.2.1. Declaring Variable-length Data

The data should be declared using the derived type hvl_t. The example below shows this.

```
TYPE(hvl_t), DIMENSION(1:2), TARGET :: wdata
```

4.2.3.2.2. Writing and Reading Variable-length Data

Writing and reading variable-length data is done in the same way as for any other derived type. This is demonstrated by the following code from the example h5ex_t_vlen_F03.f90:

```
TYPE(hvl_t), DIMENSION(1:2), TARGET :: wdata
INTEGER, DIMENSION(:), POINTER :: ptr_r
TYPE(C_PTR) :: f_ptr
….
ALLOCATE( ptr(1:2) )
ALLOCATE( ptr(1)%data(1:wdata(1)%len) )
….
DO i=1, wdata(1)%len
     ptr(1)%data(i) = wdata(1)%len - i + 1 ! 3 2 1
ENDDO
wdata(1)%p = C_LOC(ptr(1)%data(1))
…..
CALL H5Tvlen_create_f(H5T_NATIVE_INTEGER, memtype, hdferr)
f_ptr = C_LOC(wdata(1))
CALL h5dwrite_f(dset, memtype, f_ptr, hdferr)
CALL h5dvlen_reclaim_f(memtype, space, H5P_DEFAULT_F, f_ptr, hdferr)
```

Please notice that the HDF5 Fortran Library provides a new subroutine h5dvlen_reclaim_f that should be used to release allocated data buffers.

4.2.4. Enumerated Type

Reading and writing enumerations from a Fortran program is illustrated in `h5ex_t-enum_F03.f90`. This example can be found at http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/api18-fortran.html.

Fortran applications should carefully follow the steps described in the sub-sections below to assure that data is passed correctly to the HDF5 library.

4.2.4.1. Steps to Write or Read Data of an Enumerated Type

Go through these steps to write or read data of an enumerated datatype.

4.2.4.1.1. Variable Declarations

To write or read HDF5 enum data, one should use an enumerated type using ENUMERATOR as shown in the following example:

```
! Enumerated type
  ENUM, BIND(C)
     ENUMERATOR :: SOLID = 0, LIQUID, GAS, PLASMA
  END ENUM
```

The buffers with data to write or read should be declared as using integers of the KIND that corresponds to the enumerator and the TARGET attribute as shown below:

```
INTEGER(KIND(SOLID)), DIMENSION(1:dim0, 1:dim1), TARGET :: wdata ! Write buffer
INTEGER(KIND(SOLID)), DIMENSION(:,:), ALLOCATABLE, TARGET :: rdata ! Read buffer
```

4.2.4.1.2. Constructing a Memory Datatype

A memory datatype should be constructed by finding an appropriate HDF5 integer datatype and then following the standard procedure for constructing an HDF5 enum type. To find an appropriate HDF5 integer datatype, use the `h5kind_to_type` function as shown in the following example:

```
M_BASET   = h5kind_to_type(kind(SOLID), H5_INTEGER_KIND) ! Memory base type
CALL h5tenum_create_f (M_BASET, memtype, hdferr)
DO i = SOLID, PLASMA
…
     val = i
     CALL h5tenum_insert_f(memtype, TRIM(names(i+1)), val, hdferr)
….
ENDDO
```

4.2.4.1.3. Constructing a File Datatype

If the datatype in the file is different from `memtype`, special care should be taken to convert enum values before constructing the HDF5 enum datatype. Conversion is performed on each value using the `h5tconvert_f` subroutine. Please note that the enum value has to have a datatype big enough to contain a converted value. The following example may not work if the file datatype would be, for example, `H5T_STD_I64BE`.

```
INTEGER(kind(SOLID)), TARGET :: val
….
! In the file enums are 16-bit integers
CALL h5tenum_create_f (H5T_STD_I16BE, filetype, hdferr)
DO i = SOLID, PLASMA
…
      !
      ! Insert enumerated value for filetype.  We must first convert
      ! the numerical value val to the base type of the destination.
      !
      f_ptr = C_LOC(val)
      CALL h5tconvert_f (M_BASET, H5T_STD_I16BE, INT(1,SIZE_T), f_ptr, hdferr)
      CALL h5tenum_insert_f(filetype, TRIM(names(i+1)), val, hdferr)
  ENDDO
```

4.2.4.1.4. Creating, Writing, and Reading Enums

To create a dataset with an HDF5 enum datatype, use a standard `h5dcreate_f` call. Here is an example:

```
CALL h5dcreate_f(file, dataset, filetype, space, dset, hdferr)
```

After the dataset is created, write data using the new signature for the `h5dwrite_f` subroutine as shown below:

```
f_ptr = C_LOC(wdata(1,1))
CALL h5dwrite_f(dset, memtype, f_ptr, hdferr)
```

Reading is a similar process. First, find the size of the buffer to hold data and allocate it, and then use `h5dread_f` as shown:

```
  CALL h5dget_space_f(dset,space, hdferr)
  CALL h5sget_simple_extent_dims_f (space, dims, maxdims, hdferr)
  ALLOCATE(rdata(1:dims(1),1:dims(2)))
  !
  ! Read the data.
  !
  f_ptr = C_LOC(rdata(1,1))
  CALL h5dread_f(dset, memtype, f_ptr, hdferr)
```

4.3. HDF5 Fortran APIs with Callbacks

The Fortran 2003 standard allows us to implement Fortran wrappers for the HDF5 C functions that use callback functions. Those APIs are the following: `h5eget_auto_f`, `h5literate_f`, `h5literate_by_name_f`, `h5ovisit_f`, and `h5pcreate_class_f`.

The example `h5ex_g_iterate_F03.f90` shows how to use a callback function written in Fortran to iterate over the groups and their members. The module `liter_cb_mod` contains a callback Fortran function `op_func` that uses the `H5Oget_info_by_name_f` subroutine to discover the types of the objects, to discover the names of the links to them, and to print this information. The C pointer to this function is passed as a parameter to the `h5literate_f` subroutine that iterates over all objects found in the file specified by the file identifier `file`. See the code examples below.

```
MODULE liter_cb_mod
  USE HDF5
  USE ISO_C_BINDING
INTEGER FUNCTION op_func(loc_id, name, info, operator_data) bind(C)
…..
TYPE(H5O_info_t), TARGET :: infobuf
ptr = C_LOC(infobuf)
TYPE(C_PTR) :: ptr
CALL H5Oget_info_by_name_f(loc_id, name_string, ptr, status)
IF(infobuf%type.EQ.H5O_TYPE_GROUP_F)THEN
       WRITE(*,*) "Group: ", name_string
    ELSE IF(infobuf%type.EQ.H5O_TYPE_DATASET_F)THEN
……
END FUNCTION op_func
END MODULE liter_cb_mod

PROGRAM main
  USE HDF5
  USE ISO_C_BINDING
  USE liter_cb_mod
  ……..
  funptr = C_FUNLOC(op_func)
  ptr = C_LOC(info)
  CALL h5literate_f(file, H5_INDEX_NAME_F, H5_ITER_NATIVE_F, idx, funptr, ptr,
ret_value, status)
…….
END PROGRAM main
```

4.4. Backward and Forward Compatibility Issues

Fortran 2003 features do not affect the HDF5 file format or the Fortran APIs available in earlier versions of the software. Fortran applications written for versions of the library prior to HDF5 Release 1.8.8 will work without any changes with HDF5 Release 1.8.8 and later. HDF5 files written by the 1.8.8 version of the HDF5 Fortran Library using APIs introduced in HDF5 Release 1.8.8 can be read by earlier versions of the HDF5 1.8.* library.

4.5. Source Code File Structure

There were several additions to the Fortran source code file structure in the 1.8.8 release.

The source code for all Fortran APIs that require Fortran 2003 features is located in the fortran/src directory in the files with the names containing the "F03" string. For example, `H5Lff_F03.f90` contains Fortran wrappers for the H5L C interface. The source code in those files is conditionally compiled in the release when the `--enable-fortran2003` configure flag is specified along with the `--enable-fortran` flag during the HDF5 configuration step.

The Fortran test directory fortran/test contains new files with the "F03" string in their names. The files contain tests for the new APIs. As for the source, the tests are conditionally compiled in when the Fortran 2003 features are available.

The Fortran example directory fortran/examples has three new examples to illustrate Fortran 2003 features: `compound_fortran2003.f90`, `nested_derived_type.f90`, and `rwdset_fortran2003.f90`.

4.6. Fortran API Changes and Additions in Version 1.8.8

The table below shows existing Fortran functions that have been upgraded and new functions introduced in HDF5 Release 1.8.8. The upgraded functions have been changed so that they now pass a pointer to the buffer instead of passing the data buffer itself. The functions in the New Functions column have had no Fortran implementation prior to version 1.8.8. They have had a C implementation. See the "HDF5 Software Changes from Release to Release" page on the web site for more information.

API Interface	Upgraded Functions	New Functions
H5	h5open_f	
	h5close_f	
	h5check_version_f	
	h5get_libversion_f	
	h5garbage_collect_f	
	h5dont_atexit_f	
H5A	h5aread_f	
	h5awrite_f	
H5D	h5dread_f	
	h5dwrite_f	

API Interface	Upgraded Functions	New Functions
		h5dvlen_reclaim_f
H5DS		h5dsattach_scale_f
		h5dsdetach_scale_f
		h5dsget_label_f
		h5dsget_num_scales_f
		h5dsget_scale_name_f
		h5dsis_attached_f
		h5dsis_scale_f
		h5dsset_label_f
		h5dsset_scale_f
H5E		h5eset_auto_f
H5L		h5literate_by_name_f
		h5literate_f
H5O		h5ovisit_f
		h5oget_info_by_name_f
H5P	h5pset_fill_value_f	
	h5pget_fill_value_f	
	h5pset_f	
	h5pget_f	
	h5pregister_f	
	h5pinsert_f	
	h5pcreate_class_f	
		h5pset_nbit_f
		h5pset_scaleoffset_f
H5R	h5rcreate_f	
	h5rdereference_f	
	h5rget_name_f	
	h5rget_object_type_f	
H5T		h5tconvert_f
HDF5 Utility		h5offsetof
		h5kind_to_type