RFC: Update to HDF5 References

Jerome Soumagne
Quincey Koziol
Elena Pourmal

HDF5 references allow users to reference existing HDF5 objects as well as selections within datasets. The original API, however, must be extended in order to add the ability to reference attributes as well as objects in external files. Additionally, there are some inherent limitations within the current API that restrict its use with virtual object layer (VOL) connectors, which do not necessarily follow HDF5's native file format.

This document presents the design and implementation of the revised reference API. It introduces a single opaque reference type, which not only presents the advantage of hiding the internal representation of references, it also allows for future extensions to be added more seamlessly.

Revision History

Version Number	Date	Comments
v1	Jul. 12, 2018	Version 1 circulated for comment within The HDF Group.
v2	Aug. 6, 2018	Version 2 circulated for comment within The HDF Group.
v3	Aug. 10, 2018	Version 3 circulated for comment within The HDF Group.
v4	Jan. 31, 2019	Version 4 circulated for comment within The HDF Group.
v5	Feb. 6, 2019	Version 5 circulated for comment within The HDF Group.
v6	Jun. 12, 2019	Version 6 circulated for comment within The HDF Group.
v7	Aug. 15, 2019	Version 7 circulated for comment within The HDF Group.

Contents

List of Figures

Introduction

The existing HDF5 reference API currently allows users to create references to HDF5 objects (groups, datasets) and regions within a dataset. However, it presents some limitations: it defines two separate reference types `hobj_ref_t` and `hdset_reg_ref_t`; the former directly maps to an `haddr_t` type that does not allow for external references, while the latter maps to an HDF5 global heap entry, which is specific to native HDF5 and gets created and written to the file when the reference is created. This not only prevents users from creating region references when the file is opened read-only, it is also not suitable for use outside of native HDF5 files. This RFC aims at addressing these limitations by introducing a single abstract `href_t` type as well as missing reference types such as *attribute* references and *external* references (i.e., references to objects in an external file). In section 1, we review in more details the limitations within the existing API. In sections 2 and 3, we go over the required public API changes. In section 4, we present the internal implementation changes and discuss some of the remaining points that will need to be addressed.

1. Existing API

While the current API has been providing support for both object and region references, it is lacking in terms of functionality and flexibility: there is no support for attribute references; references are only valid within the container that they reference; the size of the reference types is tied to the definition of an `haddr_t` or an entry in the file's global heap, which only exists in native HDF5. With the virtual object layer (VOL) feature now integrated into HDF5 (see 4.3), it becomes a requirement to provide VOL connectors with a straightforward and flexible way of using references.

Beside the actual type definition of references, function definitions must also be revised. Firstly, the current `H5Rcreate` signature forces users to constantly pass `(H5I_INVALID_HID)` as a `space_id`, in the case where the reference type is *not* a region reference. Secondly, the extra `loc_id` parameter that is passed to all the functions becomes source of confusion when expanding to external references and, for simplicity, can be embedded within the reference type itself. Finally and as previously mentioned, because the size of region references is currently defined as the size required to encode a global heap ID, the current definition forces references to be written to the file at the time of their creation, hence preventing them to be created from a file that is opened read-only (e.g, when creating references to a file that one does not want/cannot modify).

For reference, the original API is defined below:

```
/* Note! Be careful with the sizes of the references because they should
 * really depend on the run-time values in the file.  Unfortunately, the
 * arrays need to be defined at compile-time, so we have to go with the
 * worst case sizes for them. -QAK
 */
#define H5R_OBJ_REF_BUF_SIZE        sizeof(haddr_t)

/* 4 is used instead of sizeof(int) to permit portability between the
 * Crays and other machines (the heap ID is always encoded as an int32
 * anyway).
 */
#define H5R_DSET_REG_REF_BUF_SIZE   (sizeof(haddr_t) + 4)
```

```
/* Reference types */
typedef enum H5R_type_t {
    H5R_BADTYPE     =   (-1),   /* Invalid Reference Type               */
    H5R_OBJECT,                 /* Object reference                     */
    H5R_DATASET_REGION,         /* Dataset Region Reference             */
    H5R_MAXTYPE                 /* Highest type (Invalid as true type)  */
} H5R_type_t;

/* Object reference structure for user's code
 * This needs to be large enough to store largest haddr_t on a worst case
 * machine (8 bytes currently).
 */
typedef haddr_t hobj_ref_t;

/* Dataset Region reference structure for user's code
 * (Buffer to store heap ID and index)
 * This needs to be large enough to store largest haddr_t in a worst case
 * machine (8 bytes currently) plus an int
 */
typedef unsigned char hdset_reg_ref_t[H5R_DSET_REG_REF_BUF_SIZE];

/* Prototypes */
herr_t  H5Rcreate(void *ref, hid_t loc_id, const char *name,
                  H5R_type_t ref_type, hid_t space_id);
hid_t   H5Rdereference2(hid_t obj_id, hid_t oapl_id,
                        H5R_type_t ref_type, const void *ref);
hid_t   H5Rget_region(hid_t dataset, H5R_type_t ref_type,
                      const void *ref);
herr_t  H5Rget_obj_type2(hid_t id, H5R_type_t ref_type,
                         const void *_ref, H5O_type_t *obj_type);
ssize_t H5Rget_name(hid_t loc_id, H5R_type_t ref_type,
                    const void *ref, char *name /*out*/, size_t size);
```

2. Revised API

Most notable changes are the three `H5Rcreate_X()` calls for each reference type: *object*, *dataset region* and *attribute reference*. It is also worth noting that as opposed to the previous implementation, creating a region reference no longer modifies the original file (which is particularly useful in scenarios where one wants to store references in a separate file or container). A direct consequence, however, is that each reference created must be released by a call to `H5Rdestroy()`. Other changes are self explanatory and aim at making the API simpler. Note that the `H5Rdereference()` call has now been replaced with `H5Ropen_object()`, which is more in line with the other existing `H5Ropen_X()` calls. Backward compatibility is discussed in section 3.

The revised API is described below:

```
/* Default reference buffer size. */
#define H5R_REF_BUF_SIZE        64
```

```
/**
 * Opaque reference type. The same reference type is used for object,
 * dataset region and attribute references.
 */
typedef unsigned char href_t[H5R_REF_BUF_SIZE];

/*
 * Reference types allowed.
 */
typedef enum {
    H5R_BADTYPE     =   (-1),   /* Invalid reference type           */
    H5R_OBJECT1     =     0,    /* Backward compatibility (object)  */
    H5R_DATASET_REGION1 = 1,    /* Backward compatibility (region)  */
    H5R_OBJECT2         = 2,    /* Object reference                 */
    H5R_DATASET_REGION2 = 3,    /* Region reference                 */
    H5R_ATTR            = 4,    /* Attribute Reference              */
    H5R_MAXTYPE         = 5     /* Highest type (invalid)           */
} H5R_type_t;

/* Constructors */
herr_t  H5Rcreate_object(hid_t loc_id, const char *name, href_t *ref_ptr);
herr_t  H5Rcreate_region(hid_t loc_id, const char *name, hid_t space_id,
                         href_t *ref_ptr);
herr_t  H5Rcreate_attr(hid_t loc_id, const char *name, const char *attr_name,
                       href_t *ref_ptr);
herr_t  H5Rdestroy(href_t *ref_ptr);

/* Info */
H5R_type_t H5Rget_type(const href_t *ref_ptr);
htri_t  H5Requal(const href_t *ref1_ptr, const href_t *ref2_ptr);
herr_t  H5Rcopy(const href_t *src_ref_ptr, href_t *dst_ref_ptr);

/* Dereference */
hid_t   H5Ropen_object(const href_t *ref_ptr, hid_t oapl_id);
hid_t   H5Ropen_region(const href_t *ref_ptr);
hid_t   H5Ropen_attr(const href_t *ref_ptr, hid_t aapl_id);

/* Get type */
herr_t  H5Rget_obj_type3(const href_t *ref_ptr, hid_t oapl_id,
                         H5O_type_t *obj_type);

/* Get name */
ssize_t H5Rget_file_name(const href_t *ref_ptr, char *name, size_t size);
ssize_t H5Rget_obj_name(const href_t *ref_ptr, hid_t oapl_id, char *name,
 ↪  size_t size);
ssize_t H5Rget_attr_name(const href_t *ref_ptr, char *name, size_t size);

/* Serialization */
herr_t  H5Rencode(const href_t *ref_ptr, void *buf, size_t *nalloc);
herr_t  H5Rdecode(hid_t loc_id, const void *buf, size_t *nbytes,
                  href_t *ref_ptr);
```

References can be stored and retrieved from a file by invoking the `H5Dwrite()` and `H5Dread()` functions with the following single predefined type:

```
#define H5T_STD_REF         (H5OPEN H5T_STD_REF_g)
hid_t H5T_STD_REF_g;
```

The advantage of a single type is that it becomes easier for users to mix references of different types. It is also more in line with the opaque type now defined for references. Note that when reading references back from a file, the library may, in consequence of this new design, allocate memory for each of these references. To release the memory, one must either call `H5Rdestroy()` on each of the references or, for convenience, call the new `H5Treclaim()` function on the buffer that contains the array of references (type can be compound type, array). An example is shown in appendix A.1.

As mentioned, instead of having separate routines for both vlen and reference types, we unify the existing:

```
herr_t H5Dvlen_reclaim(hid_t type_id, hid_t space_id, hid_t dxpl_id,
                       void *buf);
```

to:

```
herr_t H5Treclaim(hid_t type_id, hid_t space_id, hid_t dxpl_id, void *buf);
```

3. API Compatibility

To preserve compatibility with applications and middleware libraries that have been using the existing reference API, we keep for now the existing `H5Rcreate()`, `H5Rdereference2()`, `H5Rget_region()`, `H5Rget_obj_type2()` and `H5Rget_name()` routines, hoping that they can be moved to the deprecated API list of functions in the future.

It is important to note though that these routines only support the original reference types, noted as `H5R_OBJECT1` and `H5R_DATASET_REGION1` respectively. Any other reference type passed to these routines will return an error. For convenience and compatibility with previous versions of the library we define both `H5R_OBJECT` and `H5R_DATASET_REGION` to map to the original reference types[1].

```
/* Macros for compatibility */
#define H5R_OBJECT H5R_OBJECT1
#define H5R_DATASET_REGION H5R_DATASET_REGION1

/* Prototypes */
herr_t  H5Rcreate(void *ref, hid_t loc_id, const char *name,
                  H5R_type_t ref_type, hid_t space_id);
herr_t  H5Rget_obj_type2(hid_t id, H5R_type_t ref_type, const void *_ref,
                         H5O_type_t *obj_type);
hid_t   H5Rdereference2(hid_t obj_id, hid_t oapl_id, H5R_type_t ref_type,
                        const void *ref);
hid_t   H5Rget_region(hid_t dataset, H5R_type_t ref_type, const void *ref);
ssize_t H5Rget_name(hid_t loc_id, H5R_type_t ref_type, const void *ref,
                    char *name, size_t size);
```

[1] With the revised API, the reference type being used is internally assigned by the library and no longer controlled by the user.

```
/* Deprecated prototypes */
#ifndef H5_NO_DEPRECATED_SYMBOLS
H5G_obj_t H5Rget_obj_type1(hid_t id, H5R_type_t ref_type,
                           const void *_ref);
hid_t     H5Rdereference1(hid_t obj_id, H5R_type_t ref_type, const void *ref);
#endif /* H5_NO_DEPRECATED_SYMBOLS */
```

When creating and accessing references through these routines, users are still expected to use the following datatypes, which describe the `hobj_ref_t` and `hdset_reg_ref_t` types:

```
#define H5T_STD_REF_OBJ       (H5OPEN H5T_STD_REF_OBJ_g)
#define H5T_STD_REF_DSET_REG (H5OPEN H5T_STD_REF_DSET_REG_g)

hid_t H5T_STD_REF_OBJ_g;
hid_t H5T_STD_REF_DSET_REG_g;
```

One important aspect of these changes is to ensure that previously written data can still be readable after those revisions and that new files produced will not create any undefined behavior when used with previous versions of the library. Backward as well as forward compatibility is summarized in table 1.

Table 1 – Backward/Forward Compatibility.

	Old File Format		New File Format	
Version	Old API	New API	Old API	New API
$< 1.12.0$	No change	N/A	Datatype version bump prevents from reading unknown reference types	N/A
$\geq 1.12.0$	Read and write references through old datatypes and use `hobj_ref_t` and `hdset_reg_ref_t` types	Read and write using `H5T_STD_REF` to convert to new `href_t` type	Cannot use old API with new reference types	Can use opaque `href_t` type for all reference types

Because previous library versions do not have a way of detecting when new unknown references types are read, we have to increment the global version of the datatypes, so that early detection can be done and appropriate error be returned to the user. For versions prior to this change, the library will return an error when a datatype encountered has a version number greater than the currently supported version. Also to prevent datatype version changes in the future, all library branches are now patched to check for unknown reference types.

When reading old data with the new library version, one can either keep using the `H5T_STD_REF_OBJ` and `H5T_STD_REF_DSET_REG` datatypes, which can be queried when opening a dataset for example using `H5Dget_type()`, or use the `H5T_STD_REF` datatype, which will trigger automatic type conversion. The `H5T_STD_REF_OBJ` and `H5T_STD_REF_DSET_REG` datatypes require the use of the respective `hobj_ref_t` and `hdset_reg_ref_t` types, which can only be used with the old API functions. These types do not embed all the required information to be simply cast to an `href_t` type. When an `href_t` type is desired, the

H5T_STD_REF datatype must be used, allowing old reference data to be used with the new API[2]. A usage example is illustrated in appendix A.2.

4. Internal Design and Implementation

References touch three core pieces of the library: the actual H5R reference module, the H5T datatype module and the H5VL virtual object layer, which includes connectors.

4.1. Reference Implementation

The internal href struct describes object, region and attribute references and is defined as:

```
/* Object reference */
struct href_obj {
    char *filename;             /* File name */
    haddr_t addr;               /* Object address */
};

/* Region reference */
struct href_reg {
    struct href_obj obj;        /* Object reference */
    H5S_t *space;               /* Selection */
};

/* Attribute reference */
struct href_attr {
    struct href_obj obj;        /* Object reference */
    char *name;                 /* Attribute name */
};

/* Generic reference type */
struct href {
    union {
        struct href_obj  obj;   /* Object reference                */
        struct href_reg  reg;   /* Region reference                */
        struct href_attr attr;  /* Attribute Reference             */
    } ref;
    hid_t loc_id;               /* Cached location identifier      */
    hsize_t encode_size;        /* Cached encoding size            */
    H5R_type_t type;            /* Reference type                  */
    char unused[16];            /* Padding                         */
};
```

This implementation can be summarized in table 2. We also cache additional information such as the location attached to the reference (usually a file ID) and the buffer size required to encode a reference.

[2]We only allow for conversion between old and new references. Conversion from new references to old references is not allowed.

Table 2 – Internal representation of reference types.

Reference Type	Representation
Object	`File name + object address`
Dataset Region	`Object + selection`
Attribute	`Object + attribute name`

4.2. Datatype Implementation

There are now three different HDF5 datatypes for references: `H5T_STD_REF_OBJ`, `H5T_STD_REF_DSET_REG` and `H5T_STD_REF`. `H5T_STD_REF_OBJ` and `H5T_STD_REF_DSET_REG` remain unchanged and it is important to note that `H5T_STD_REF_DSET_REG` remains native-only. The `H5T_STD_REF` datatype that represents the `href_t` type has a memory representation that is different from its disk representation. Therefore it requires special handling and conversion through the datatype conversion module.

4.2.1. Memory Representation

The *serialized* representation of an `href_t` type is described in table 3.

Table 3 – Encoded representation of reference types.

Reference Type	Encoded Representation
Object	`Version + type + flags + (serialized VOL info + filename string) + object address`
Dataset Region	`Object + serialized selection`
Attribute	`Object + attribute name string`

Note that if the flag `H5R_IS_EXTERNAL` is set, additional information is encoded such as the filename, VOL info. This extra information is not needed if the reference is going to be stored in the same destination file as the file it references.

4.2.2. Disk Representation

The *serialized* representation of an `href_t` must be contained in a variable length buffer[3]. Therefore, writing or reading a reference to a file uses a similar code path as for the variable length types of HDF5. In the case of native HDF5[4], a serialized `href_t` must be stored in the file's global heap (using the `H5HG_insert()`) routine at the time of datatype conversion. What gets actually stored at write time in the allocated disk space is the encoded ID that points to the entry in the global heap (i.e., the address of the collection + the entry index within the collection, see figure 1).

[3] Object references that reference the same file they are stored in are fixed size and can be stored with minimal overhead
[4] We can discuss here the new blob API for VOL connectors once it is ready to be used.

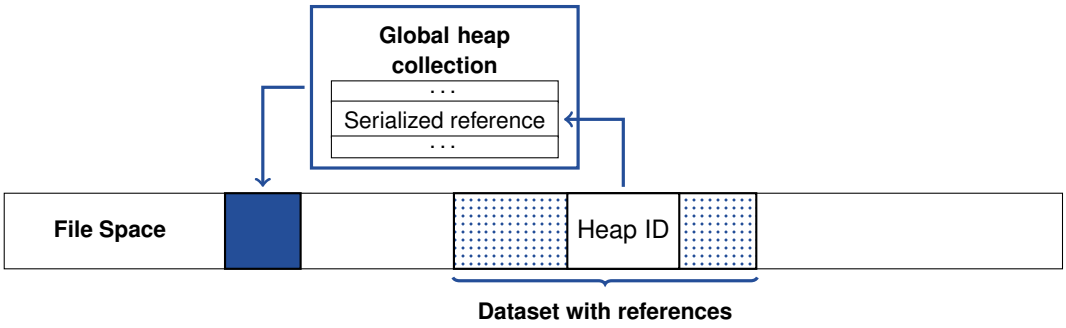

Figure 1 – Disk representation of an HDF5 variable length reference.

4.3. VOL Implementation

The current definition of VOL structures implied the definition of reference callbacks by the VOL connector. With this new revised implementation, we simplify and refactor what is handed off to the VOL connector so that only the actual object address or token (`haddr_t`) gets provided to the reference API. This implies that region references and attribute references are no longer VOL connector specific since the serialized form of HDF5 selections and attribute names is independent of the VOL being used.

Are therefore removed from the current VOL definition:

```
/* types for object GET callback */
typedef enum H5VL_object_get_t {
-    H5VL_REF_GET_NAME,                 /* object name                */
-    H5VL_REF_GET_REGION,               /* dataspace of region        */
-    H5VL_REF_GET_TYPE                  /* type of object             */
     H5VL_ID_GET_NAME                  /* object name, for hid_t */
} H5VL_object_get_t;

/* types for object SPECIFIC callback */
typedef enum H5VL_object_specific_t {
     H5VL_OBJECT_CHANGE_REF_COUNT,     /* H5Oincr/decr_refcount   */
     H5VL_OBJECT_EXISTS,               /* H5Oexists_by_name       */
     H5VL_OBJECT_VISIT,                /* H5Ovisit(_by_name)      */
-    H5VL_REF_CREATE,                  /* H5Rcreate               */
     H5VL_OBJECT_FLUSH,                /* H5{D|G|O|T}flush        */
     H5VL_OBJECT_REFRESH               /* H5{D|G|O|T}refresh      */
} H5VL_object_specific_t;

/* types for different ways that objects are located in an
 * HDF5 container */
typedef enum H5VL_loc_type_t {
     H5VL_OBJECT_BY_SELF,
     H5VL_OBJECT_BY_NAME,
     H5VL_OBJECT_BY_IDX,
     H5VL_OBJECT_BY_ADDR,
-    H5VL_OBJECT_BY_REF
```

```
} H5VL_loc_type_t;

- struct H5VL_loc_by_ref {
-     H5R_type_t ref_type;
-     const void *_ref;
-     hid_t lapl_id;
- };
```

The only capabilities that are therefore needed from the VOL connector to support references are:

1. the ability to lookup an object by its name and return an `haddr_t` token that represents the object address;
2. the ability to retrieve an object's name by its address;
3. the ability to retrieve an object's type by its address.

Are added to the VOL object class definition:

```
/* types for object GET callback */
 typedef enum H5VL_object_get_t {
+    H5VL_OBJECT_GET_NAME,               /* object name              */
+    H5VL_OBJECT_GET_TYPE                /* object type              */
 } H5VL_object_get_t;

/* types for object SPECIFIC callback */
 typedef enum H5VL_object_specific_t {
     H5VL_OBJECT_CHANGE_REF_COUNT,      /* H5Oincr/decr_refcount    */
     H5VL_OBJECT_EXISTS,                /* H5Oexists_by_name        */
+    H5VL_OBJECT_LOOKUP,                /* Lookup object by name    */
     H5VL_OBJECT_VISIT,                 /* H5Ovisit(_by_name)       */
     H5VL_OBJECT_FLUSH,                 /* H5{D|G|O|T}flush         */
     H5VL_OBJECT_REFRESH                /* H5{D|G|O|T}refresh       */
 } H5VL_object_specific_t;
```

4.4. Outstanding Issues

1. If a link is deleted, the actual reference will be invalid. Should we increment the refcount on the object when a reference gets created? or should we just let the object reference be invalid if the link gets deleted?

4.5. Library Wrappers

4.5.1. Fortran

No particular issue to report but must be done.

4.5.2. Java

No particular issue to report but must be done.

4.5.3. Tools

Tools must be updated to support the new `H5T_STD_REF` datatype. However, there is currently no issue to report with that update.

A. Usage Examples

A.1. External References

The example below illustrates the use of the new API with files that are opened read-only. Created references to the objects in that file are stored into a separate file, and accessed from that file, without the user explicitly opening the original file that was referenced.

```
#define NDIMS    1        /* Number of dimensions */
#define BUF_SIZE 4        /* Size of example buffer */
#define NREFS    1        /* Number of references */

int
main(void)
{
    hid_t file1, dset1, space1;
    hsize_t dset1_dims[NDIMS] = { BUF_SIZE };
    int dset_buf[BUF_SIZE];

    hid_t file2, dset2, space2;
    hsize_t dset2_dims[NDIMS] = { NREFS };
    href_t ref_buf[NREFS] = { 0 };
    H5O_type_t obj_type;
    int i;

    for (i = 0; i < BUF_SIZE; i++)
        dset_buf[i] = i;

    /* Create file with one dataset and close it */
    file1 = H5Fcreate("file1.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
    space1 = H5Screate_simple(NDIMS, dset1_dims, NULL);
    dset1 = H5Dcreate2(file1, "dataset1", H5T_NATIVE_INT, space1, H5P_DEFAULT,
                       H5P_DEFAULT, H5P_DEFAULT);
    H5Dwrite(dset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, dset_buf);
    H5Dclose(dset1);
    H5Sclose(space1);
    H5Fclose(file1);

    /* Create reference to dataset1 in "file1.h5" */
    file1 = H5Fopen("file1.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
    H5Rcreate_object(file1, "dataset1", &ref_buf[0]);
    H5Fclose(file1);

    /* Store reference in dataset in separate file "file2.h5" */
```

```
    file2 = H5Fcreate("file2.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
    space2 = H5Screate_simple(NDIMS, dset2_dims, NULL);
    dset2 = H5Dcreate2(file2, "references", H5T_STD_REF, space2, H5P_DEFAULT,
                       H5P_DEFAULT, H5P_DEFAULT);
    H5Dwrite(dset2, H5T_STD_REF, H5S_ALL, H5S_ALL, H5P_DEFAULT, ref_buf);
    H5Dclose(dset2);
    H5Sclose(space2);
    H5Fclose(file2);
    H5Rdestroy(&ref_buf[0]);

    /* Read reference from "file2.h5" */
    file2 = H5Fopen("file2.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
    dset2 = H5Dopen2(file2, "references", H5P_DEFAULT);
    H5Dread(dset2, H5T_STD_REF, H5S_ALL, H5S_ALL, H5P_DEFAULT, ref_buf);
    H5Dclose(dset2);
    H5Fclose(file2);

    /* Access reference and read dataset data */
    assert(H5Rget_type(&ref_buf[0]) == H5R_OBJECT2);
    H5Rget_obj_type3(&ref_buf[0], H5P_DEFAULT, &obj_type);
    assert(obj_type == H5O_TYPE_DATASET);
    dset1 = H5Ropen_object(&ref_buf[0], H5P_DEFAULT);
    H5Dread(dset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, dset_buf);
    H5Dclose(dset1);
    H5Rdestroy(&ref_buf[0]);

    for (i = 0; i < BUF_SIZE; i++)
        assert(dset_buf[i] == i);

    return 0;
}
```

A.2. Backward Compatibility and New API

The example below illustrates the use of the new API with a file that was written using the old-style reference API, showing how one can take advantage of the automatic type conversion from old reference type to new reference type.

```
#define NDIMS    1        /* Number of dimensions */
#define NREFS    1        /* Number of references */
#define BUF_SIZE 4        /* Size of example buffer */

int
main(void)
{
    hid_t file1, dset1, space1;
    hsize_t dset1_dims[NDIMS] = { BUF_SIZE };
    int dset_buf[BUF_SIZE];

    hid_t dset2, space2;
    hsize_t dset2_dims[NDIMS] = { NREFS };
```

```
    hobj_ref_t ref_buf[NREFS] = { 0 };
    href_t new_ref_buf[NREFS] = { 0 };
    H5O_type_t obj_type;
    int i;

    for (i = 0; i < BUF_SIZE; i++)
        dset_buf[i] = i;

    /* Create file with one dataset and close it */
    file1 = H5Fcreate("file1.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    space1 = H5Screate_simple(NDIMS, dset1_dims, NULL);
    dset1 = H5Dcreate2(file1, "dataset1", H5T_NATIVE_INT, space1, H5P_DEFAULT,
                       H5P_DEFAULT, H5P_DEFAULT);
    H5Dwrite(dset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, dset_buf);
    H5Dclose(dset1);
    H5Sclose(space1);

    /* Create reference to dataset1 */
    H5Rcreate(&ref_buf[0], file1, "dataset1", H5R_OBJECT, H5I_INVALID_HID);

    /* Store reference in separate dataset */
    space2 = H5Screate_simple(NDIMS, dset2_dims, NULL);
    dset2 = H5Dcreate2(file1, "references", H5T_STD_REF_OBJ, space2,
                       H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
    H5Dwrite(dset2, H5T_STD_REF_OBJ, H5S_ALL, H5S_ALL, H5P_DEFAULT, ref_buf);
    H5Dclose(dset2);
    H5Sclose(space2);
    H5Fclose(file1);

    /* Read reference from file */
    file1 = H5Fopen("file1.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
    dset2 = H5Dopen2(file1, "references", H5P_DEFAULT);
    H5Dread(dset2, H5T_STD_REF, H5S_ALL, H5S_ALL, H5P_DEFAULT, new_ref_buf);
    H5Dclose(dset2);

    /* Access reference and read dataset data */
    assert(H5Rget_type(&new_ref_buf[0]) == H5R_OBJECT2);
    H5Rget_obj_type3(&new_ref_buf[0], H5P_DEFAULT, &obj_type);
    assert(obj_type == H5O_TYPE_DATASET);
    dset1 = H5Ropen_object(&new_ref_buf[0], H5P_DEFAULT);
    H5Dread(dset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, dset_buf);
    H5Dclose(dset1);
    H5Rdestroy(&new_ref_buf[0]);

    for (i = 0; i < BUF_SIZE; i++)
        assert(dset_buf[i] == i);

    return 0;
}
```

B. Reference Manual

B.1. H5Rcreate_object

Synopsis:

```
herr_t      H5Rcreate_object(hid_t loc_id, const char *name, href_t *ref_ptr);
```

Purpose:

Creates an object reference.

Description:

`H5Rcreate_object` creates a reference pointing to the object named `name` located at `loc_id`. The parameters `loc_id` and `name` are used to locate the object.

Parameters:

`hid_t loc_id`	IN: Location identifier
`const char *name`	IN: Name of object
`href_t *ref_ptr`	OUT: Pointer to reference

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.2. H5Rcreate_region

Synopsis:

```
herr_t      H5Rcreate_region(hid_t loc_id, const char *name, hid_t space_id,
                             href_t *ref_ptr);
```

Purpose:

Creates a region reference.

Description:

`H5Rcreate_region` creates the reference, `ref_ptr`, pointing to the region represented by `space_id` within the object named `name` located at `loc_id`. The parameters `loc_id` and `name` are used to locate the object. The parameter `space_id` identifies the dataset region that a dataset region reference points to.

Parameters:

`hid_t loc_id`	IN: Location identifier
`const char *name`	IN: Name of object
`hid_t space_id`	IN: Dataspace identifier
`href_t *ref_ptr`	OUT: Pointer to reference

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.3. H5Rcreate_attr

Synopsis:

```
herr_t      H5Rcreate_attr(hid_t loc_id, const char *name,
                           const char *attr_name, href_t *ref_ptr);
```

Purpose:

Creates an attribute reference.

Description:

`H5Rcreate_attr` creates the reference, `ref_ptr`, pointing to the attribute named `attr_name` and attached to the object named `name` located at `loc_id`. The parameters `loc_id` and `name` are used to locate the object. The parameter `attr_name` is used to locate the attribute within the object.

Parameters:

`hid_t loc_id`	IN: Location identifier
`const char *name`	IN: Name of object
`const char *attr_name`	IN: Name of attribute
`href_t *ref_ptr`	OUT: Pointer to reference

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.4. H5Rdestroy

Synopsis:

```
herr_t      H5Rdestroy(href_t *ref_ptr);
```

Purpose:

Closes a reference.

Description:

Given a reference, `ref_ptr`, to an object, region or attribute attached to an object, `H5Rdestroy` releases allocated resources from a previous create call.

Parameters:

`href_t *ref_ptr` IN: Pointer to reference

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.5. H5Rget_type

Synopsis:

```
H5R_type_t H5Rget_type(const href_t *ref_ptr);
```

Purpose:

Retrieves the type of a reference.

Description:

Given a reference, `ref_ptr`, `H5Rget_type` returns the type of the reference.

Valid returned reference types are:

`H5R_OBJECT2`	Object reference version 2
`H5R_DATASET_REGION2`	Region reference version 2
`H5R_ATTRIBUTE`	Attribute reference

Note that `H5R_OBJECT1` and `H5R_DATASET_REGION1` can never be associated to an `href_t` reference and can therefore never be returned through that function.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to query

Returns:

Returns a valid reference type if successful; otherwise returns `H5R_UNKNOWN`.

B.6. H5Requal

Synopsis:

```
htri_t      H5Requal(const href_t *ref1_ptr, const href_t *ref2_ptr);
```

Purpose:

Determines whether two references are equal.

Description:

`H5Requal` determines whether two references point to the same object, region or attribute.

Parameters:

`const href_t *ref1_ptr`	IN: Pointer to reference to compare
`const href_t *ref2_ptr`	IN: Pointer to reference to compare

Returns:

Returns a positive value if the references are equal. Returns 0 if the references are not equal. Returns a negative value when the function fails.

B.7. H5Rcopy

Synopsis:

```
herr_t      H5Rcopy(const href_t *src_ref_ptr, href_t *dst_ref_ptr);
```

Purpose:

Copies an existing reference.

Description:

`H5Rcopy` creates a copy of an existing reference.

Parameters:

`const href_t *src_ref_ptr`	IN: Pointer to reference to copy
`href_t *dst_ref_ptr`	OUT: Pointer to output reference

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.8. H5Ropen_object

Synopsis:

```
hid_t      H5Ropen_object(const href_t *ref_ptr, hid_t oapl_id);
```

Purpose:

Opens the HDF5 object referenced.

Description:

Given a reference, `ref_ptr`, to an object, a region in an object or an attribute attached to an object, `H5Ropen_object` opens that object and returns an identifier.

The parameter `oapl_id` is an object access property list identifier for the referenced object. The access property list must be of the same type as the object being referenced, that is a group or dataset property list.

The object opened with this function should be closed when it is no longer needed so that resource leaks will not develop. Use the appropriate close function such as `H5Oclose` or `H5Dclose` for datasets.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to open
`hid_t oapl_id`	IN: Valid object access property list identifier

Returns:

Returns identifier of referenced object if successful; otherwise returns a negative value.

B.9. H5Ropen_region

Synopsis:

```
hid_t      H5Ropen_region(const href_t *ref_ptr);
```

Purpose:

Sets up a dataspace and selection as specified by a region reference.

Description:

`H5Ropen_region` creates a copy of the dataspace of the dataset pointed to by a region reference, `ref_ptr`, and defines a selection matching the selection pointed to by `ref_ptr` within the dataspace copy.

Use `H5Sclose` to release the dataspace identifier returned by this function when the identifier is no longer needed.

Parameters:

`const href_t *ref_ptr` IN: Pointer to region reference to open

Returns:

Returns a valid dataspace identifier if successful; otherwise returns a negative value.

B.10. H5Ropen_attr

Synopsis:

```
hid_t      H5Ropen_attr(const href_t *ref_ptr, hid_t aapl_id);
```

Purpose:

Opens the HDF5 attribute referenced.

Description:

Given a reference, `ref_ptr`, to an attribute attached to an object, `H5Ropen_attr` opens the attribute attached to that object and returns an identifier.

The parameter `aapl_id` is an attribute access property list identifier for the referenced attribute.

The attribute opened with this function should be closed with `H5Aclose` when it is no longer needed.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to attribute reference to open
`hid_t aapl_id`	IN: Valid attribute access property list identifier

Returns:

Returns a valid attribute identifier if successful; otherwise returns a negative value.

B.11. H5Rget_obj_type3

Synopsis:

```
herr_t      H5Rget_obj_type3(const href_t *ref_ptr, hid_t oapl_id,
                             H5O_type_t *obj_type);
```

Purpose:

Retrieves the type of object that an object reference points to.

Description:

Given a reference, `ref_ptr`, `H5Rget_obj_type3` retrieves the type of the referenced object in `obj_type`. The referenced object type, or the type of the referenced object, is the type of the object that the reference points to. When the function completes successfully, it returns one of the following valid object type values (defined in `H5Opublic.h`):

`H5O_TYPE_GROUP`	Object is a group
`H5O_TYPE_DATASET`	Object is a dataset
`H5O_TYPE_NAMED_DATATYPE`	Object is a named datatype

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to query
`hid_t oapl_id`	IN: Valid object access property list identifier
`H5O_type_t *obj_type`	OUT: Type of referenced object

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.12. H5Rget_file_name

Synopsis:

```
ssize_t    H5Rget_file_name(const href_t *ref_ptr, char *name, size_t size);
```

Purpose:

Retrieves the file name for a referenced object.

Description:

`H5Rget_file_name` retrieves the file name for the object, region or attribute reference pointed to by `ref_ptr`. Up to `size` characters of the name are returned in `name`; additional characters, if any, are not returned to the user application. If the length of the name, which determines the required value of `size`, is unknown, a preliminary `H5Rget_file_name` call can be made. The return value of this call will be the size of the file name. That value can then be assigned to `size` for a second `H5Rget_file_name` call, which will retrieve the actual name.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to query
`char *name`	IN/OUT: A buffer to place the file name of the reference
`size_t size`	IN: The size of the name buffer

Returns:

Returns the length of the name if successful. Otherwise returns a negative value.

B.13. H5Rget_obj_name

Synopsis:

```
ssize_t     H5Rget_obj_name(const href_t *ref_ptr, hid_t oapl_id, char *name,
                            size_t size);
```

Purpose:

Retrieves the object name for a referenced object.

Description:

`H5Rget_obj_name` retrieves the object name for the object, region or attribute reference pointed to by `ref`. Up to `size` characters of the name are returned in `name`; additional characters, if any, are not returned to the user application. If the length of the name, which determines the required value of `size`, is unknown, a preliminary `H5Rget_obj_name` call can be made. The return value of this call will be the size of the object name. That value can then be assigned to `size` for a second `H5Rget_obj_name` call, which will retrieve the actual name. If there is no name associated with the object identifier or if the name is `NULL`, `H5Rget_obj_name` returns the size of the name buffer (the size does not include the `NULL` terminator).

If `ref` is an object reference, `name` will be returned with a name for the referenced object. If `ref` is a dataset region reference, name will contain a name for the object containing the referenced region. If `ref` is an attribute reference, name will contain a name for the object the attribute is attached to. Note that an object in an HDF5 file may have multiple paths if there are multiple links pointing to it. This function may return any one of these paths.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to query
`hid_t oapl_id`	IN: Valid object access property list identifier
`char *name`	IN/OUT: A buffer to place the object name of the reference
`size_t size`	IN: The size of the name buffer

Returns:

Returns the length of the name if successful, returning 0 (zero) if no name is associated with the identifier. Otherwise returns a negative value.

B.14. H5Rget_attr_name

Synopsis:

```
ssize_t    H5Rget_attr_name(const href_t *ref_ptr, char *name, size_t size);
```

Purpose:

Retrieves the attribute name for a referenced object.

Description:

`H5Rget_attr_name` retrieves the attribute name for the attribute reference pointed to by `ref`. Up to `size` characters of the name are returned in `name`; additional characters, if any, are not returned to the user application. If the length of the name, which determines the required value of `size`, is unknown, a preliminary `H5Rget_attr_name` call can be made. The return value of this call will be the size of the attribute name. That value can then be assigned to `size` for a second `H5Rget_attr_name` call, which will retrieve the actual name.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to query
`char *name`	IN/OUT: A buffer to place the attribute name of the reference
`size_t size`	IN: The size of the name buffer

Returns:

Returns the length of the name if successful. Otherwise returns a negative value.

B.15. H5Rencode

Synopsis:

```
herr_t      H5Rencode(const href_t *ref_ptr, void *buf, size_t *nalloc);
```

Purpose:

Encodes a reference into a buffer.

Description:

Given a reference `ref_ptr`, `H5Rencode` marshalls the memory representation of a reference into a buffer, suitable for transmission. Using this representation, a reference can be reconstructed using `H5Rdecode` to return a new reference handle (`href_t`) for this reference.

A preliminary `H5Rencode` call can be made to find out the size of the buffer needed. This value is returned as `nalloc`. That value can then be assigned to `nalloc` for a second `H5Rencode` call, which will be used for the actual encoding of the reference.

If the library finds out that `nalloc` is not big enough, it simply returns the size of the buffer needed through `nalloc` without encoding the provided buffer.

Parameters:

`const href_t *ref_ptr`	IN: Pointer to reference to query
`void *buf`	IN/OUT: Buffer for the object to be encoded into
`size_t *nalloc`	IN: The size of the allocated buffer
	OUT: The size of the buffer needed

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

B.16. H5Rdecode

Synopsis:

```
herr_t      H5Rdecode(hid_t loc_id, const void *buf, size_t *nbytes,
                      href_t *ref_ptr);
```

Purpose:

Decodes a reference from a buffer.

Description:

Given a marshalled reference representation in a buffer, `H5Rdecode` reconstructs the HDF5 reference and returns a new reference handle for it. The marshalled representation is created by `H5Rencode`. User is responsible for passing in the right buffer of the appropriate `nbytes` size.

The reference returned by this function can be released with `H5Rclose` when it is no longer needed so that resource leaks will not develop.

Parameters:

`hid_t loc_id`	IN: Location identifier
`const void *buf`	IN: Buffer for the reference to be decoded
`size_t *nbytes`	IN: The size of the buffer to decode
	OUT: The number of bytes decoded
`href_t *ref_ptr`	OUT: Pointer to decoded reference

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.