
Note to readers:
Compatibility Option #4 is being implemented.
-- Posted by Frank Baker, July 31, 2007

API Compatibility Strategies for HDF5

Quincey Koziol
April 13, 2007

"Compatibility means deliberately repeating other people's mistakes"
- David Wheeler, Cambridge University

"All problems in computer science can be solved by another layer of indirection" - Butler Lampson, Xerox PARC

Background:

API compatibility has been a difficult problem for the HDF5 library over the course of its development. On the one hand, new features need to be added and bugs fixed, and the "best" solution for those modifications may be to change an existing API routine. On the other hand, we would like to make upgrading to later versions of the library as easy as possible for existing application developers. And on the third hand, we would like the library's interface to be as coherent and easy to understand for new application developers as possible.

What This Document Does & Doesn't Cover:

This document describes API compatibility requirements, problems and solutions. It does not address file format compatibility issues. As general

background, here's our policy on file format compatibility for the HDF5 library:

"Each released HDF5 library will read all existing HDF5 files, which are produced by itself or any earlier release. Although each release may contain features that require additions and/or changes to the HDF5 file format, by default each release will write out files that conform to a 'maximum compatibility' principle. That is, files are written with the earliest version of the file format that can correctly describe the information, as opposed to using the latest version defined.

This provides the best forward compatibility by allowing the maximum number of applications built with older versions of the HDF5 library to read files produced with later library releases. If library features are used that require new file format features, or if the application requests that the library write out a later version of the file format, the files produced may not be readable by older versions of the HDF5 library."

HDF5 API Compatibility Goal:

Our goal should be to make life as easy as possible for application developers that use the HDF5 library, while still maintaining as cohesive and reasonably-sized set of API routines as possible. We should look to serve both the existing users of the HDF5 library as well as future users.

Example API Change for Discussion:

In the HDF5 1.8.0 release, we would like to change the API routine for creating a group in an HDF5 file from its current signature, which is closely tied to the older format for storing groups:

```
  hid_t H5Gcreate(hid_t loc_id, const char *name, size_t size_hint);
```

to take a more flexible set of parameters:

```
  hid_t H5Gcreate(hid_t loc_id, const char *name, hid_t gcpl_id, hid_t
gapl_id);
```

However, since this API routine is used in many applications, just changing the parameters without any thought for existing application code would require those application developers to either change all their existing code (to use the 1.8.0 release) or stick with the existing 1.6.x branch of library releases.

The rest of this document uses the changes to the group creation routine as an example for how the change to HDF5 API routines affect user applications.

Compatibility Options:

Several compatibility options are described here, using the changes to the H5Gcreate() API routine as their example.

- Option #1: "Just do it" (with apologies to Nike :-)
  - Action: just change the API routine and don't worry about application compatibility.
  - Pros:
    - Easy for us - keeps library testing and code maintenance low.
    - Applications that update to the latest version of the library always use the "best" version of the feature in the library.
  - Cons:
    - All existing applications that use a changed routine will break, requiring changes to their source code.
    - Very difficult for applications to maintain compatiblity with more than one release branch of the HDF5 library.

        o - Ramifications:
            o - Lots of people won't or can't afford to change their
                application code, and so won't ever migrate to later
releases
                of the library.
            o - We will get a reputation for not caring about existing
users.
                (sometimes an attitude that open source developers have
as they
                chase the "bleeding edge" of "cool" technology and
programming
                techniques and forget that other people actually use the
code
                they are writing)

    o - Option #2: "The same as it ever was..."   (with apologies to
the
            Talking Heads :-)

        o - Action: change the API routine in the latest release, but
provide
            a configure flag to enable the old version of the API
routine.
            (i.e. --enable-compat... flag)   [This is what we have done
in
            previous major HDF5 releases (1.2.x => 1.6.x)]

        o - Pros:
            o - Supports existing applications by allowing them to use
the
                latest release, with some trade-offs.
            o - Pushes people toward later versions of the library by
making
                the new version of the API routine the default.

        o - Cons:
            o - Either have "all new" or "all old" API routines. An
application
                can't use some of the new APIs and some of the old APIs
at the
                same time.
            o - More difficult for us to maintain. We have to maintain
and
                test two separate API configurations.
            o - Still difficult for applications to maintain
compatibility with
                multiple releases of the HDF5 library - they are just
"tricking"

a later version of the library into looking like a previous
version.
- The HDF5 releases have only maintained compatibility code for
one major release, so applications have to be changed
eventually. (i.e. v1.6.x was compatible with v1.4.x, but
v1.8.0 would be only compatible with v1.6.x - applications
currently using the v1.4.x API set will have to upgrade to at
least the v1.6.x API set, if they want to use the 1.8.x
releases)

- Ramifications:
- Lots of people won't use the _features_ in the latest release
of the library.
- People can't claim that we aren't listening, but we're not
helping an _awful_ lot either...

- Option #3: "just pile it on" (there's a library growing deep and
wide... :-)

- Action: leave all existing API routines from previous version(s)
of the library and add new routines, with new features. Possibly
add configure flag to disable old routines (--without-deprecated)
Eventually, we might decide to drop API routines that have been
deprecated for a long time.

- Pros:
- Supports existing applications by allowing them to use the
latest release, without changing their code.
- Allows old and new API routines to be mixed in application's
code.

- Cons:
- Lots of API routines! We already have a lot of API routines
and this option would make things worse.

            o - Since we're leaving old routines around, we've got to come up
                with a new name for each changed API, leaving us with routine
                names either like "H5Gcreate2, H5Gcreate3, etc." or
                "H5Gcreate_foo, H5Gcreate_bar, etc.".
            o - Incoherent programming API - it will be difficult for
                application programmers to know which API routines to use.
                (i.e. the latest/best routines for the H5G API could be called
                H5Gcreate3/H5Gopen2/H5Gclose4, etc.)
            o - Lots of tests for us to write, to make certain that old and
                new API routines all work correctly.

        o - Ramifications:
            o - We are listening to and supporting our users: People will have
                a smooth upgrade path, but eventually the # of publicly visible
                API routines will be out of control and the current set of
                "best" routines difficult to understand, especially for new
                application developers.

    o - Option #4: "something old, something new"   (something borrowed,
            something blue. :-)

        o - Action: Define a "version macro" for each public API routine and
            public data structure. Use those macros to map the "best"
            (i.e. original) API names to the most current version of each API
            routine, but allow users to override those choices on both a global
            and an individual basis (i.e. the macro for H5Gcreate maps to
            H5Gcreate2 by default, but it's possible for user to easily remap
            it to H5Gcreate1 if desired). Possibly add a configure flag to
            disable old routines entirely (--without-deprecated-apis). Also
            define a "release version macro" which maps all the API routines to
            the versions they had for a particular major release (i.e. a

macro
            that allows the HDF5 library API routines to look the same
way they
            did for the 1.6.x releases or the 1.4.x releases, etc).

            [In some ways, this is a combination of options #2 & #3]

            [These changes would take care of C API compatibility,
FORTRAN 9X
            and C++ API compatibility can probably be taken care of with
            parameter overloading].

        o - Pros:
            o - Supports existing applications by allowing them to use
the
                latest release, without changing their code.
            o - Allows old and new API routines to be mixed in
application's
                code. (i.e they can call H5Gcreate1 and H5Gcreate2
"directly",
                as well as use the H5Gcreate wrapper)
            o - Pushes applications to later releases. By default, the
wrapper
                will point to the new version of the API routine.
            o - More coherent programming API, the "best" names will
continue
                to do the "best" thing.
            o - Keeps the # of publicly visible API routines more in
check.
            o - May allow us to add other API wrappers, for performance
                testing, etc. (Similar to how MPICH library is
implemented)

        o - Cons:
            o - Requires more knowledge from application developers than
                option #3, but less overall work than option #2 or #1.
            o - Lots of tests for us to write, to make certain that old
and
                new API routines all work correctly. Along with tests
on the
                mapped API names.
            o - Lots of documentation changes.
            o - Needs very good "how do I update my application/library
to
                the latest release" document, that shows application
                developers their options and has good examples. We'll
still
                get lots of questions to the helpdesk, probably...
            o - Routine name remapping with macros could make debugging

harder
for application programmers: they think they are calling a
different routine name than they actually do.

[We could add a configure option to disable the macro remapping
and make the latest version of a routine actually be called
the "best" name during the HDF5 library build & install.
However, this is problematic if we make the definition of
H5Gcreate2 go away and the application is explicitly calling
it, for example...]

[Possibly help the situation by having an API routine that an
application could call to display values of "version macros",
as that routine "saw" them when compiled? Still not very
helpful in the case of multiple application source code modules
(that call HDF5 routines) being compiled with different version
macro settings... :-/]

[This is a hard problem to solve, we may just have to advise
developers of the issue and assume that they will build all the
code that calls HDF5 routines with the same version macros
defined]

- o - Using macros this way could possibly remap text in a C/C++
  user application that was unrelated to the HDF5 library.

  [This is probably pretty minimal, since it's unlikely that user
  applications will have symbols in their code that are identical
  to our API routine/data structure names]

- o - Ramifications:
  - o - We are listening to and supporting our users: People will have
    a smooth upgrade path and hopefully the # of publicly

visible
API routines will be better controlled and stay more coherent.

Our Chosen Path:
==========

Put simply, we believe that option #4 is the most beneficial to current and future HDF5 application developers at the [relatively minor] expense of some extra testing and configuration work by the HDF5 development team, and is the path we are planning to embark upon, beginning with the 1.8.0 release. The choices for application developers and a sketch of the implementation for option #4 is elaborated below.

Ideally, if we implement option #4, any HDF5 application ever written will be able to link against the HDF5 1.8.0 or later release (possibly using a macro to choose a particular release version of the public APIs). This will greatly ease the process of upgrading to later HDF5 releases for application developers, who will gain several benefits from doing so:

- Improvements in the HDF5 library's execution time, memory use, or platform support.
- Bug fixes found in later releases that weren't ported back to earlier release branches.
- Transparent ability to read HDF5 files containing file format changes from later releases of the HDF5 library.

Option #4 Application Developer Choices:
==========

With this option, an application that used H5Gcreate would "automatically" migrate to using H5Gcreate2 when they recompiled against the 1.8.0 release and generate errors during either the compile or link phase. This will require the application developer's attention to determine a course of action

for
porting the application to the HDF5 release's changed APIs.

The developers would have several choices when compiling with an HDF5
release that used this API compatibility option:

- o - Upgrade their code. We should provide guidelines for changing
  an existing application to use new versions of each API routine.

  In this case, the easiest change for a developer will be to change
  all H5Gcreate calls to drop the "size_hint" parameter and add
  two H5P_DEFAULT parameters, and add a H5P_DEFAULT parameter to all
  H5Gopen calls.

- o - Change their code to stick with a version, by changing all
  calls to H5Gcreate to H5Gcreate1, H5Gopen to H5Gopen1, etc.

  This should be a simple global search and replace in their
  source code.

- o - Build and install the HDF5 library with the "1.6 compatibility"
  configure flag, which would define the "H5_USE_16_API" macro in the
  header files installed and thus make all the API version macros map
  to the 1.6 API set. Then, the application code wouldn't need to be
  modified right away. However, each API routine could be
  individually changed in the application to use the later version of
  that API and the application developer could redefine that API's
  macro to the later version, leaving all the other routines back at
  the older version of the API, enabling a developer to incrementally
  migrate to the latest release.

  For example, the developer uses the configure flag and installs
  the library, getting his application to work. Then, they start
  converting each HDF5 API routine to the latest version.

When they've changed all the H5Gcreate calls to use the H5Gcreate2 version, they would re-compile their application with H5Gcreate version macro defined to the value "2", overriding the mapping to the older version in the API compatibility header file, but leaving the H5Gopen macro map to H5Gopen1.

Of course, installing the library this way forces all the other users of the library to default to the 1.6 API set. This is addressed in the next developer option.

o - Install the HDF5 library with its default settings (and new API versions). Then, define the H5_USE_16_API macro when the application is built, switching the application's "view" of the HDF5 library to the 1.6 API set. Again, the application can incrementally convert to using newer versions of each API routine individually, by defining each API routine's version macro to a later version, while keeping all the other API routines at the older API version and continuing to use the H5_USE_16_API macro until all the API routine calls in the application were converted to the latest versions and the H5_USE_16_API release version macro could be dropped.

o - If only a few API routines are causing problems, the HDF5 library can be installed with its default settings and just the macros for the API routines that are causing problems can be defined by the application to the older versions, until the application is updated. In this case, if calls to H5Gcreate were the only API routine causing a problem, the developer would just define the H5Gcreate version macro to "1", but leave the rest of the version

macros alone.

The HDF5 library could also be compiled with a configure flag (--disable-deprecated-symbols) or used with a macro (H5_NO_DEPRECATED_SYMBOLS)
that disabled deprecrated API routines, allowing a smaller library installation
(with the configure option) or easy way to detect use of deprecated API routines while still allowing other applications to use them (with the macro).

Option #4 Implementation:
==========

Since option #1 doesn't require any work and we're fairly familiar with
how options #2 & #3 would work, this section goes into detail about how option
#4 could be implemented. Again, the H5Gcreate (and H5Gopen) API routines
are used as an example.

Option #4 uses "version macros" to control how the "best" API routines
are mapped onto the actual API implementation routines. These macros are
defined for both individual routines and entire sets of routines for a release
series. This could be implemented in a way similar to the following, in a
header file:

==========

```
/* Allow the user to choose to map all API routines with a single macro
 *      or configuration flag (using the 1.6.x versions of the API routines
 *      shown here), but also allow them to override single routine names if
 *      they desire.
 */
#ifdef H5_USE_16_API
#if !defined(H5Gcreate_vers)
#define H5Gcreate_vers 1
#endif
#if !defined(H5Gopen_vers)
#define H5Gopen_vers 1
```

```
#endif
#endif /* H5_USE_16_API */

/* Map the "best" API names to latest version of the routine, if user
has not
 *      already caused that API name to be defined.
 */
#if !defined(H5Gcreate_vers) || H5Gcreate_vers == 2
#define H5Gcreate H5Gcreate2
#elif H5Gcreate_vers == 1
#define H5Gcreate H5Gcreate1
#else /* H5Gcreate_vers */
#error "H5Gcreate_vers set to unknown value"
#endif /* H5Gcreate_vers */

#if !defined(H5Gopen_vers) || H5Gopen_vers == 2
#define H5Gopen H5Gopen2
#elif H5Gopen_vers == 1
#define H5Gopen H5Gopen1
#else /* H5Gopen_vers */
#error "H5Gopen_vers set to unknown value"
#endif /* H5Gopen_vers */
```

==========

Any data structures that are passed to an API routine which changed, would
be aliased to the correct struct, based on the mapping for the routine name.

For example, H5Gget_objinfo() takes a struct called H5G_stat_t. If a
different version of H5Gget_objinfo() was created, with the implementation of
option #4 outlined above, there would be an old version of the API routine
called H5Gget_objinfo1 and taking a struct called H5G_stat1_t and a new version
of it called H5Gget_objinfo2 and taking a struct called H5G_stat2_t. The
definition of H5G_stat_t would be remapped depending on the mapping for
H5Gget_objinfo, like so:

```
#if H5Gget_objinfo_vers == 2
#define H5G_stat_t H5G_stat2_t
#else
#define H5G_stat_t H5G_stat1_t
#endif
```

So, if an application developer remaps the H5Gget_objinfo API routine, the
H5G_stat_t definition will change as well.

Since the macro definitions for each mapped API routine are so regular,
we can use the same "generative programming" technique that we use for creating
the error API public header files: write a perl script that parses a text file
and produces the appropriate header(s), with all the macro definitions. This
will reduce the complexity of defining these macros and make them simpler for
library developers to modify.

[The error API perl script is in bin/make_err and the text file it parses is
src/H5err.txt, generating the H5Einit.h, H5Epubgen.h, H5Eterm.h and H5Edefin.h
header files, in the 'src' directory. The 'make_err' script is called by the
bin/reconfigure script, allowing the generated header files to be shipped with
the library and avoids requiring a perl interpreter during the build and
install steps at users' sites.

This is somewhat similar to how the src/H5Tinit.c source module is generated,
but that code is generated at build time on each machine with a C program
(src/H5detect.c), since it's specific to the machine being built on.]

Sample Prototype Implementation of Option #4:

The following four files compose a small but complete example of how this
option can be implemented.

The 'lib.c' file is compiled with the following command line:

```
gcc -g lib.c -o lib.o -c
```

or with this command line, if deprecated APIs are to be removed at library
build time:

```
        gcc -g lib.c -D H5_NO_DEPRECATED_SYMBOLS -o lib.o -c

    The "application" file, 'appl.c', can be compiled with several
options,
depending on the API mapping desired.  For a default mapping of the API
routines, this command line can be used:
        gcc -g appl.c lib.o -o appl

    To map all the API routines to their v1.6.x equivalents, this
command line
can be used:
        gcc -g appl.c -D H5_USE_16_API lib.o -o appl

    To map all the API routines to their v1.6.x equivalents, but map
H5Gcreate
to version 2 of the API routine, this command line can be used:
        gcc -g appl.c -D H5_USE_16_API -D H5Gcreate_vers=2 lib.o -o appl

    This command line demonstrates making deprecated API routines
"invisible"
at application build time:
        gcc -g appl.c -D H5_NO_DEPRECATED_SYMBOLS lib.o -o appl


====================== lib.c ======================

#include <stdio.h>

#include "public.h"

/* Remove old API routines, if library is built without them. */
#ifndef H5_NO_DEPRECATED_SYMBOLS
int H5Gcreate1(create1_type x)
{
    printf("first version of create\n");
}

int H5Gopen1(open1_type x)
{
    printf("first version of open\n");
}
#endif /* H5_NO_DEPRECATED_SYMBOLS */

int H5Gcreate2(create2_type x)
{
    printf("second version of create\n");
}

int H5Gopen2(open2_type x)
```

```
{
    printf("second version of open\n");
}
```

```
==================== compat.h ====================

/* Issue error if contradicting macros have been defined. */
#if defined(H5_USE_16_API) && defined(H5_NO_DEPRECATED_SYMBOLS)
#error "Can't choose old API versions when deprecated APIs are disabled"
#endif /* defined(H5_USE_16_API) && defined(H5_NO_DEPRECATED_SYMBOLS) */

/* If a particular "global" version of the library's interfaces is
chosen,
 *      set the versions for the API routines affected.
 *
 * Note: If an application has already chosen a particular version for
an
 *      API routine, the individual API version macro takes priority.
 */
#ifdef H5_USE_16_API
#if !defined(H5Gcreate_vers)
#define H5Gcreate_vers 1
#endif
#if !defined(H5Gopen_vers)
#define H5Gopen_vers 1
#endif
#endif /* H5_USE_16_API */

/* Choose the correct version of each API routine, defaulting to the
latest
 *      version of each API routine.  The "best" name for API
parameters/data
 *      structures that have changed definitions is also set.  An error
is
 *      issued for specifying an invalid API version.
 */
#if !defined(H5Gcreate_vers) || H5Gcreate_vers == 2
#define H5Gcreate H5Gcreate2
#define create_type create2_type
#elif H5Gcreate_vers == 1
#define H5Gcreate H5Gcreate1
#define create_type create1_type
#else /* H5Gcreate_vers */
#error "H5Gcreate_vers set to unknown value"
#endif /* H5Gcreate_vers */

#if !defined(H5Gopen_vers) || H5Gopen_vers == 2
#define H5Gopen H5Gopen2
#define open_type open2_type
```

```
#elif H5Gopen_vers == 1
#define H5Gopen H5Gopen1
#define open_type open1_type
#else /* H5Gopen_vers */
#error "H5Gopen_vers set to unknown value"
#endif /* H5Gopen_vers */

==================== public.h ====================

#include "compat.h"

typedef struct {
    int i;
    float f;
} create2_type;

typedef struct {
    int i;
    float f;
} open2_type;

/* Allow for older API routines and data structures to be completely
compiled
 *      out at library build time, or rendered "invisible" at
application build
 *      time.
 */
#ifndef H5_NO_DEPRECATED_SYMBOLS

typedef struct {
    int i;
} create1_type;

typedef struct {
    int i;
} open1_type;

extern int H5Gcreate1(create1_type x);
extern int H5Gopen1(open1_type x);
#endif /* H5_NO_DEPRECATED_SYMBOLS */

/* Prototypes for latest versions of the API routines */
extern int H5Gcreate2(create2_type x);
extern int H5Gopen2(open2_type x);

==================== appl.c ====================

/*
 * This code demonstrates the feasibility of the API compatibility
```

```
version
 *      macro implementation.
 */

#include <stdio.h>
#include "public.h"

int main()
{
    /* Mapped data structures */
    create_type x;
    open_type y;

    /* Hard-wired data structures */
#ifndef H5_NO_DEPRECATED_SYMBOLS
    create1_type x1;
#endif /* H5_NO_DEPRECATED_SYMBOLS */
    create2_type x2;

    /* Mapped API routine calls */
    H5Gcreate(x);
    H5Gopen(y);

    /* Hard-wired API routine calls */
#ifndef H5_NO_DEPRECATED_SYMBOLS
    H5Gcreate1(x1);
#endif /* H5_NO_DEPRECATED_SYMBOLS */
    H5Gcreate2(x2);

    return(0);
}
```

Command-Line Options for Configure Script With Option #4:
==

The configure script shipped with the HDF5 distribution will need to add several command-line options that allow application developers to control the API version macros defined at the HDF5 software's build time. The following options are proposed:

--disable-deprecated-symbols
: This option maps all versioned symbols to the latest version available, removing all previous versions from both application visibility and compiled library code.

It is an error to define --disable-deprecated-symbols and choose a
default release (with the --with-default-api-version configure option)
earlier than the current release.

Default setting is to enable deprecated versioned symbols.

--with-default-api-version=[v10|v12|v14|v16|v18]
This option maps the all versioned symbols to version defined by
the _last_ release of a particular release branch. (v10 corresponds
to the 1.0 release branch, v12 corresponds to the 1.2 release branch,
etc). Versioned symbols introduced in a later release branch are
still available.

It is an error to define --disable-deprecated-symbols and choose a
default release (with the --with-default-api-version configure option)
earlier than the current release.

Default setting is to map all versioned symbols to the version defined
by the current release.

Individual API routine name versioning is not controlled with configure
script options, they must be retargeted with version macros defined at
application build time.