Writing HDF5 tests

Jordan Henderson

This document is intended to be a general guide for writing HDF5 test programs.

Contents

1. Introduction

Writing HDF5 tests can be a time-consuming process that is made worse when newly-written tests pass on a development machine but fail or exhibit problematic behavior in other places and on other platforms where HDF5 is tested. This document outlines some of the more common problems test authors may encounter and is intended to be a general guide for writing HDF5 tests.

2. Testing frameworks

The following sections include details about the various testing frameworks in HDF5 and when to use each one.

2.1. 'Testframe' framework

HDF5 includes a testing framework contained within the `test/testframe.h` and `test/testframe.c` files that has basic functionality such as skipping tests, controlling the amount of output from tests, limiting the amount of time that tests run for and more (refer to Appendix A for an overview of the framework). While the functionality from this testing framework is used for a handful of older HDF5 tests, many new HDF5 tests end up being written as a simple `main()` function that runs tests directly. However, it is recommended that test authors integrate their tests with the 'testframe' framework going forward, if possible. The ability to prevent tests from running too long is especially important for parallel and multi-threaded tests, where forgetting to enable a test timer can mean that a test runs for a very long time if it hangs. This is also especially important for HPC environments, where the job queues used typically have a relatively short maximum running time (often around 30 minutes - 1 hour) and a hanging test will prevent getting results from other tests as they won't have a chance to run. For the time being, the `testframe.h` header file also includes `h5test.h`, so the entire serial testing framework infrastructure is available to tests that integrate with the 'testframe' framework.

Recent work has shown that integrating with this testing framework can have some rough edges, especially due to the fact that each test is added **before** the command-line arguments are parsed. However, integration with this testing framework brings several benefits and is more maintainable than the current approach to writing new HDF5 tests completely from scratch. Refer to Appendix D.1 and D.2 for examples of skeleton test programs integrating with this testing framework.

2.2. 'H5test' framework

HDF5 also includes a set of utility macros and functions contained within the `test/h5test.h` and `test/h5test.c` files for use by tests (refer to Appendix B for an overview of the framework). HDF5 tests should include the `h5test.h` header if they need access to the relevant utility macros and functions. HDF5 tests which don't integrate with the 'testframe' testing framework typically use this framework, but 'testframe' tests may also make use of it.

2.3. 'Testpar' framework

HDF5 also includes a set of utility macros and functions contained within the `testpar/testpar.h` and `testpar/testpar.c` files for use by parallel tests (refer to Appendix C for an overview of the framework). In general, all parallel tests should include the `testpar.h` header. This header also includes the `h5test.h` and `testframe.h` headers, so the entirety of HDF5's testing framework infrastructure is currently available to parallel tests.

3. General considerations

The following are general considerations that test authors should keep in mind when writing HDF5 tests.

3.1. Coupling of test programs and sub-tests

It is best to avoid any coupling between test programs, as well as sub-tests in a test program, where possible. This is often a problem when one test depends on a previous test having created a file. When tests are coupled in this manner and an earlier test fails, it can be difficult to diagnose the problems in the tests that run later on in the sequence of tests. It also reduces testing coverage until issues in the earlier tests/sub-tests are fixed and adds complexity to the build system code (where dependencies between tests have to be explicitly added). A good practice is to make use of helper functions to factor out duplicated code between tests and sub-tests to avoid this coupling.

3.2. Suppress HDF5 error stacks only as needed

By default, HDF5 will print out a stack of error messages to `stderr` when an API function fails. Before a refactoring, the 'testframe' framework previously would disable HDF5's error stack mechanism and rely on tests to explicitly print out the HDF5 error stack as needed. This adds maintenance overhead for test authors and also makes debugging of tests more difficult. Test programs should generally leave HDF5's error stacks enabled (unless disabled explicitly via a command-line argument or some other mechanism) and add error stack suppression calls around operations that are expected to fail and cause error stack output. Such operations can be surrounded by the `H5E_BEGIN_TRY` and `H5E_END_TRY` macros, as in:

```
H5E_BEGIN_TRY
{
    status = ...;
}
H5E_END_TRY
if (status < 0)
    error;
```

Tests should capture some form of return value from the operation inside the macros and then check the return value outside of the macros to avoid skipping over the logic in the `H5E_END_TRY` macro. Otherwise, the HDF5 error stack may become disabled for the rest of the test program's execution, potentially hiding error stacks from other test failures later on.

3.3. Limit testing time by TestExpress value

HDF5's testing framework functionality includes an internal variable that a test program can query the value of and use to determine whether it should skip some portion of testing in order to complete faster. While only a few of HDF5's tests currently pay attention to this variable's value, it is especially important for parallel and multi-threaded tests to use this value in conjunction with an application timer in order to safeguard against deadlocks / livelocks and other such issues that can cause tests to run for longer than intended. While CMake can kill off misbehaving tests after a certain amount of time has passed, this timeout value (`DART_TESTING_TIMEOUT`) can be changed[1] (and sometimes is, for certain HPC machines), allowing misbehaving parallel/multi-threaded tests to consume more testing time unnecessarily. Further, Autotools (which doesn't implement this functionality) will need to continue being supported for the foreseeable future. The value of this variable can be retrieved by calling either `GetTestExpress()` (for 'testframe' tests) or `h5_get_testexpress()` (for 'h5test' tests).

The meanings for the different values of the `TestExpress` variable are as follows:

- 0 - Tests should take as long as necessary
- 1 - Tests should take no more than 30 minutes
- 2 - Tests should take no more than 10 minutes
- 3 (or higher) - Tests should take no more than 1 minute (default in HDF5)

Note that the limit imposed by the value of the `TestExpress` variable is intended to be the limit on the total runtime of an individual test program, even if that test program consists of multiple sub-tests. This implies that test programs should:

- Set an `alarm(2)`-like timer at the start of the test program to ensure that the test program exits in a timely fashion according to the value of `TextExpress`. Note that HDF5's testing framework has the `TestAlarmOn()` / `TestAlarmOff()` functions, but these currently use a compile-time value for the timer that can only be overridden by an environment variable (`HDF5_ALARM_SECONDS`). They also depend on the availability of the `alarm(2)` function, which shouldn't generally be an issue at this point but may be problematic for MinGW builds as commented on in `testframe.c`.
- Keep track of the number of sub-tests that will be performed and divide the total allowed runtime among sub-tests to ensure that they all run to some degree of completion, even for more restrictive values of `TestExpress`. Note that some margin may need to be included in this calculation so that each sub-test can run to some degree of completion (while possibly spilling over their timeout by a small amount) and the test program can cleanup without running into the test alarm.
- Engineer sub-tests to accept and gracefully handle specified timeout values, taking care to check against the timeout value relatively frequently so as to not steal runtime from other sub-tests. Setting some form of flag when a sub-test or the entire test program has exceeded an allowed timeout value would also be a reasonable approach to give time for test cleanup. Alternatively, POSIX timers could be used to simplify test logic a bit.
- Engineer parallel/multi-threaded tests to adjust workload to try to accommodate the `TestExpress` runtime limit as the number of processes/threads involved increases

[1] Note that it may also **need** to be changed, as the default value is a 20 minute timeout for a test and, as noted further on, it may be desirable to allow 30 minutes for a test

3.4. Be VOL connector and VFD aware

Test authors should be aware that their tests may end up being run with a VOL connector or Virtual File Driver that is different from the default. This can result in test failures when testing behavior that is specific to the native HDF5 VOL connector and/or default sec2 VFD. Tests should make use of the `h5_using_native_vol()` utility function from `h5test.c`, as well as any capability flags queried from a VOL connector with `H5Pget_vol_cap_flags()`, in order to skip testing of functionality that non-native VOL connectors may not support. Tests should make use of the various `h5_XXX_driver_XXX()` utility functions from `h5test.c` in order to skip testing of functionality that non-sec2 file drivers may not support. Tests should make use of the `h5_fixname()` utility function from `h5test.c` in order to generate file names that make sense for different file drivers.

3.5. Make use of 'testframe' verbosity levels

For tests that integrate with the 'testframe' testing framework, test authors are encouraged to make use of the framework's verbosity level settings. This allows tests to include printing out of debugging information and other useful output without needing to recompile the test. When a problem occurs, the test can then be debugged by simply passing a specific value to the `-verbose` command-line option, making the debugging process simpler and preventing debug code from going stale.

3.6. Temporary files

Test programs should offer the ability to skip deleting of temporary files that were created by the test when it exits. It is very useful for debugging purposes to have those temporary files around when a test is failing. For 'testframe' tests, this can be achieved by checking the return value of the `GetTestCleanup()` function before calling `H5Fdelete()` or similar. For 'h5test' tests, this can be achieved by making use of the `h5_cleanup()` function to cleanup test files. Both functions are affected by whether the `HDF5_NOCLEANUP` environment variable is defined. If it is defined, `GetTestCleanup()` will return `false` and `h5_cleanup()` will avoid cleaning up test files, as long as the test program previously called one of the `h5_fixname(_xxx)` functions.

3.7. Test initialization functions

HDF5 tests that integrate with the 'testframe' testing framework should make sure to call the function `TestInit()` as soon as possible after HDF5 is initialized. Tests that don't integrate with the 'testframe' testing framework should make sure to call the function `h5_test_init()` as early as possible in the `main()` function. These functions set up testing infrastructure state such as the `TestExpress` functionality. For serial tests, these calls should generally come before any use of HDF5. For parallel tests, these calls should come before any use of HDF5, but also **after** a call to `MPI_Init()`.

3.8. Flush output stream before aborting testing

If an HDF5 test uses a function, such as `abort()` or `MPI_Abort()`, to abruptly terminate program execution on errors rather than gracefully handling them, it should be sure to flush the output stream (e.g., `stderr` or `stdout`) used to ensure the output isn't lost.

3.9. Scope of functions and global variables

All functions and global variables in a test should be marked as `static` unless there is reason for them to have wider scope. This prevents some oddities that can occur when different tests have similarly-named functions or variables and is a best practice in general. If a test author is considering a wider scope for a function or variable for use in another test, it is likely that the function or variable should simply be moved into one of the testing frameworks as appropriate.

4. Serial tests

4.1. Error checking and handling in serial tests

Serial HDF5 tests can make use of the `TESTING()`, `H5_FAILED()`, `TEST_ERROR()`, etc. macros for error handling if they include the `h5test.h` header. Tests that are part of the `testhdf5` test program should avoid this and instead only use the macros from the `testhdf5.h` header to prevent problems that can occur when mixing and matching the sets of macros.

5. Parallel tests

5.1. Flexibility in number of MPI processes

Parallel HDF5 tests should be written to work correctly with any number of MPI processes. While the test may not necessarily use all of those processes, parallel HDF5 tests are ran with varying numbers of MPI processes, depending on the system, from 2 and up. If a test is written to only run with a specific number of processes, this reduces test coverage on different machines.

Parallel tests should also give consideration towards whether strong or weak scaling is appropriate for the amount of testing being performed. Several of HDF5's parallel tests exhibit weak scaling, where the problem size is increased as the number of processes involved increases. While fine for small numbers of processes or small problem size growth, some of these tests grow quickly with the number of MPI processes and can be unreasonable to run for larger numbers of processes. Parallel tests that exhibit strong scaling tend to work well for CI; those that exhibit weak scaling need a little extra thought towards problem size growth rate.

5.2. Error checking and handling in parallel tests

Parallel HDF5 tests often only make use of the `VRFY()` macro from the `testpar.h` header for error handling. Parallel tests can make use of the `TESTING()`, `H5_FAILED()`, `TEST_ERROR()`, etc. macros

if they include the `h5test.h` header, but these macros don't account for multiple processes and also use problematic `goto`-style error handling. Therefore, these macros are usually only used from a single process in parallel tests, if at all.

HDF5 tests are often written to use `goto`-style error handling, where the test skips to a labelled section to perform cleanup when a part of the test fails. In a parallel HDF5 test, this can result in a hang if the part of the test that failed was a collective MPI operation, or if any collective MPI operation occurs in any following part of the test, and the failure occurred on only one MPI process or a subset of the MPI processes involved. When a failure occurs in a parallel HDF5 test, that failure should be communicated to all the MPI processes involved using some form of MPI collective communication operation so all the processes can cleanup and exit the test. Since it isn't generally feasible to predict which MPI processes will fail ahead of time, writing resilient parallel HDF5 tests usually involves something like an `MPI_Allreduce()` call after each collective operation to share a status value among all MPI processes before checking to see whether any of the processes failed. However, many of these types of calls in a parallel test may result in a lot of communication overhead, so it's generally encouraged to try to minimize the number of collective operations that occur.

5.3. Output from parallel tests

Without careful consideration, parallel HDF5 tests can accidentally be written in a manner that causes console output to be printed from every running MPI process. This generally results in interleaved output and can make it difficult to debug issues, especially when the number of MPI processes is large. To prevent this, parallel test programs should take one of the following approaches:

- Only print output from a single MPI process choosing a "leader". Most often, tests simply pick the MPI process that has a rank value of 0. The rank value of an MPI process can be retrieved by calling the `MPI_Comm_rank()` function on an MPI Communicator object (typically `MPI_COMM_WORLD`). Output from other MPI processes can be sent to the process with rank value 0 using various MPI functions.
- Coordinate MPI processes using collective communication operations such that only one MPI process will be able to print output at a time.
- Print output from MPI processes into separate log files, where each file is solely owned by a single MPI process and is named based on some algorithmic strategy. While best left for debugging purposes only, this approach can be very useful for developers to reason about problems in parallel tests.

The `MAINPROCESS` macro (defined in `testpar.h`) is a convenient way of handling the first option, as in:

```
if (MAINPROCESS)
    printf("Output from MPI process 0\n");
```

In order to use this macro, a variable called `mpi_rank` which contains the MPI rank value of the current process will need to be defined in the scope where the macro is used.

Note that any output from the 'testframe' testing framework, such as the printing out of test names or information, will only occur in the MPI process with the rank value of 0 and so does not needed to be guarded against by test programs integrated with this testing framework.

5.4. Random values

If using random values in a parallel test program where the values will be generated on multiple MPI processes during test execution, a few points should be kept in mind:

- If an initial seed value is required and the test program is designed for all the MPI processes to generate the same sequence of random values, the test program should usually generate the seed value on a single MPI process and then broadcast that value to other MPI processes with, e.g. `MPI_Bcast()`. While this may not always be necessary, it's a good practice to avoid cases where MPI processes can have different seed values. For example, when the seed value is calculated based off of the `time()` function, MPI processes may get different results for the seed value depending on process startup time.
- If the test is designed for all the MPI processes to generate the same sequence of random values, test authors should be careful to ensure that random value generation functions that have hidden state, such as `rand()`, are called by all the MPI processes in the same fashion. Due to this, it can often be better to simply generate a random value on a single MPI process and then broadcast that value to other MPI processes with, e.g. `MPI_Bcast()`.

5.5. Avoiding unintended pre-initialization of HDF5

Parallel HDF5 attempts to create a destructor attribute on the MPI Communicator object `MPI_COMM_SELF` when the library initializes. This is to ensure that proper shutdown of the library occurs during calls to `MPI_Finalize()` so that the library doesn't unintentionally make use of MPI objects after MPI has been finalized. For example, this can often occur when a parallel HDF5 application exits and has left the ID open for a File Access Property List that has had an MPI Communicator or Info object associated with it. If the application has not explicitly called `H5close()` when exiting, but has already called `MPI_Finalize()`, HDF5 will eventually call `MPI_Comm_free()` or `MPI_Info_free()` after MPI has been finalized, possibly resulting in an application crash.

If MPI has not been initialized with a call to `MPI_Init()` before HDF5 is initialized, the library will be unable to register its destructor attribute on `MPI_COMM_SELF` and will be unable to properly shut down the library in these cases. To ensure this does not happen, parallel HDF5 tests should avoid any actions that would cause HDF5 to initialize before MPI is initialized. This means avoiding:

- calling `H5open()` or any public HDF5 API function that initializes the library before `MPI_Init()`. Few public HDF5 API functions don't initialize the library so it's often safest to avoid use of any public HDF5 API functions before calls to `MPI_Init()`.
- making use of any public HDF5 macros that use the `H5open()` public API function through use of the `H5OPEN` macro. This includes assigning `hid_t` variables to predefined datatypes such as `H5T_NATIVE_INT` or making use of macros such as `H5F_ACC_TRUNC`.

To be explicit about ordering, it is also recommended for parallel tests to directly call `H5open()` as soon as possible after `MPI_Init()`.

6. Multi-threaded tests

6.1. Integrating with HDF5's 'testframe' testing framework

As developers will already be dealing with the difficulty of multi-threading in regards to both library and test code development, the best approach to writing multi-threaded tests will be to integrate with HDF5's 'testframe' testing framework (that is, include and use the functionality from `testframe.h` / `testframe.c`), simplifying portions of testing code and making multi-threaded tests consistent with each other. As issues are found with adapting a multi-threaded test into this testing framework, changes to the framework should be made to make the process easier for other tests in the future. Refer to Appendix D.3 for an example of a skeleton for a multi-threaded test program that integrates with the 'testframe' testing framework.

6.2. Allowing for a configurable number of threads

Any multi-threaded HDF5 test program should allow for specifying a maximum number of threads to be used within that program. While sub-tests of the test program may end up using less than the specified number of threads, they should generally be designed to be flexible and also not impose a minimum required number of threads (i.e., should be able to run with a single thread if need be). After integrating with the 'testframe' testing framework, multi-threaded test programs will have the command-line option `-maxthreads` that allows specifying the maximum number of threads the program can spawn in addition to the main thread for the program. The test program can then call the `GetTestMaxNumThreads()` utility function to query this value and make adjustments at run time. When a multi-threaded test program is running with a `TestExpress` value that indicates it should only perform smoke checks, it is recommended that the test program spawn a random number of threads up to the maximum allowed in order to have better testing coverage of multi-threaded functionality.

6.3. Error handling in multi-thread tests

Similar to HDF5's parallel tests, handling errors in multi-threaded HDF5 tests will require careful design from test authors to prevent hangs that starve out other tests[2]. When something fails in a multi-threaded HDF5 test, that failure may need to be collectively communicated so that other threads can cleanup and exit the test. Multi-threaded test authors are encouraged to:

1. Minimize the number of synchronization points, where possible
2. Avoid HDF5's typical `goto`-style error handling at synchronization points and instead capture a success or failure value that can be communicated between threads[3] to ensure they collectively proceed or fail

[2] This is especially important for running CI on HPC machines, where jobs are typically run on debug queues that generally have around 30 minute allowances before a job is killed.

[3] Using mutual exclusion or atomic instructions to increment a shared variable on failure, then waiting on the result with a `pthread_barrier_wait()` call seems to be a rudimentary but sufficient approach

6.4. Output from multi-threaded tests

Similar to parallel HDF5 tests, multi-threaded HDF5 tests will also have issues with interleaved output without careful consideration. Multi-threaded tests should:

- Only print output from a single thread after choosing a ”leader”. A reasonable strategy would be to algorithmically pick a leader based on the ”thread ID” that HDF5 chooses for the thread and stores in that thread’s specific/local storage.
- Use locking primitives or POSIX’s `flockfile()` to prevent interleaving of output. Ideally this locking would eventually be integrated into the testing framework and exposed by utility functions to simplify the writing of testing code. Note that, even with this locking, output can become interleaved if it occurs across multiple `printf()`-like function calls; tests will need to either group all output from a thread into a single lock-unlock cycle (likely at the end of the test), or have some synchronization point between ”rounds” of output from threads.
- Print output from threads into separate log files, where each file is solely owned by a single thread and is named based on some algorithmic strategy. While best left for debugging purposes only, this approach can be very useful for developers to reason about problems in multi-threaded tests.

Note that any output from the ’testframe’ testing framework, such as the printing out of test names or information, should not occur in multiple threads simultaneously and so does not needed to be guarded against by test programs integrated with this testing framework.

6.5. Organization and differentiation of tests

Historically, integration tests have often not been written for new HDF5 features, with the vast majority of the library’s tests consisting of regression and unit/functionality tests. This often leads to surprising behavior in the future when different HDF5 modules interact. Due to the generally non-deterministic behavior when testing multi-threaded code, regression and unit tests alone will likely be insufficient for finding issues in multi-threaded HDF5; differences in application timing across platforms alone can be enough for specific problematic behaviors to not be reproduced. The most useful tests will rather be unit and integration tests that focus on testing the functionality of a specific module of HDF5 (e.g., H5I, H5P, etc.), as well as testing the interactions between different modules.

Therefore, this document proposes that some level of organization should be introduced to the HDF5 library’s testing directory structure as multi-threaded tests are added. At a minimum, a new directory[4] should be created under the library’s `test` directory; one may also be created under the `testpar` directory for future multi-threaded parallel tests if necessary. Under this new directory should be further sub-directories that separate unit tests from integration tests, where the unit tests sub-directory has multiple sub-directories, one for each HDF5 module tested, and the integration tests directory has sub-directories based on the specific ”feature” (or combination of modules) being tested. The hope is that this organization structure will encourage developers to:

- Place their unit tests in the ”correct” location, rather than spending time trying to determine if there is an existing test file their test should be placed in, or if a new test file should be created

[4] Suppose this directory is called ’threads’ for the time being

- Be able to glance at the integration tests directory and quickly determine which modules need to be considered when it comes to cross-module interactions

Over time, this organization could also be applied to the library's existing serial tests as well. Figure 1 is a visualization of what this would look like.

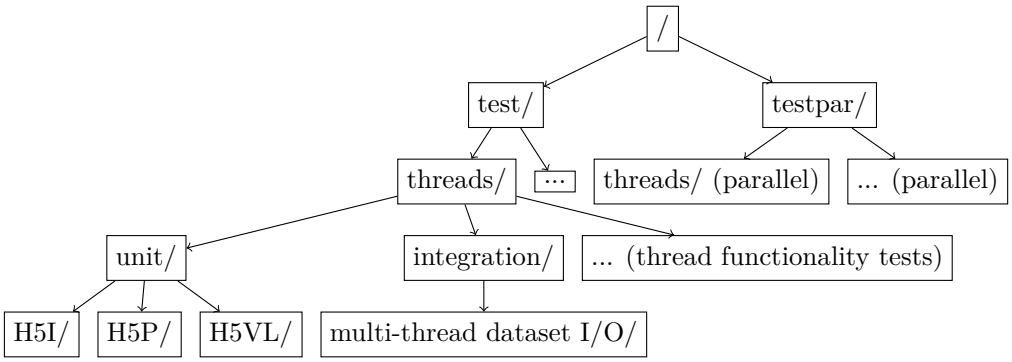

Figure 1 – Organization of HDF5 multi-threaded tests

7. Further considerations

7.1. Parallel tests

7.1.1. MPI implementations

After a parallel HDF5 test program is reasonably complete, it is always a good idea to run that test program against multiple MPI implementations to catch problems that may occur from differences in implemented functionality, values of limits, etc. between the MPI implementations. For parallel HDF5, this usually means a test program should, at a minimum, be tested against recent versions of MPICH and OpenMPI.

7.2. Multi-threaded tests

7.2.1. `TestExpress` values

The `TextExpress` variable is currently set to a default value of 3 in order to facilitate quicker CI testing of HDF5. While this chosen value may be acceptable for testing multi-threaded HDF5 functionality for changes in GitHub PRs, this value will be too restrictive for general testing of multi-threaded HDF5. If it is desired that multi-threaded HDF5 functionality be tested against changes in GitHub PRs, multi-threaded tests could be split into a separate GitHub CI action to minimize their effects by running them in parallel alongside other testing. However, this would likely still impose a 30 minute time limit as the general consensus seems to be that tests which take longer than this tend to slow down the development process when PRs are actively being created. More exhaustive testing will likely need to occur outside of GitHub, as actions are limited to 6 hours of runtime unless being executed on a machine not owned by GitHub. An effective cadence for exhaustive

testing would need to be determined so that developers can be informed reasonably quickly when new issues are introduced to multi-threaded HDF5.

7.2.2. Facilitating debugging of multi-thread issues

When debugging an issue that occurs while testing multi-threaded HDF5, it is advantageous to stop the process as immediately as possible in order to prevent program state from progressing. To this end, it has been proposed that `assert()` statements should be used to achieve this effect while running from within a debugger. To be the most effective, this would likely require that `assert()` statements be included in both the HDF5 testing framework and the HDF5 library itself. While the former should be fairly easy to accomplish with effort from a test program's author and also by updating some of the testing macros in `h5test.h`, the latter is of more concern for general use of HDF5. While `assert()` statements will only have an effect in debug builds of HDF5, past experience has shown that users employ these types of builds for general use. It is proposed that the "developer" build mode[5], which was added to HDF5's CMake build code, should also be added to the library's Autotools build code (provided that it remains around for the foreseeable future) and these `assert()` statements should only be active with a "developer" build of the library. This allows for much more freedom to introduce heavy-handed debugging techniques, while not affecting users of HDF5 in general.

Also note that, after some research on the topic[6], it appears that whether or not an `assert()` statement stops at a useful place during debugging, or even stops at all, varies among platforms. However, compiler features can help out a bit here, with the following given as an example that might be useful:

```
#ifdef DEBUG
#  if __GNUC__
#    define assert(c) if (!(c)) __builtin_trap()
#  elif _MSC_VER
#    define assert(c) if (!(c)) __debugbreak()
#  else
#    define assert(c) if (!(c)) *(volatile int *)0 = 0
#  endif
#else
#  define assert(c)
#endif
```

These compiler directives are better suited for debugging purposes as they generate trap-like instructions and generally force a much more abrupt process termination as desired. Testing across the compilers/platforms that HDF5 currently supports would be needed to determine if this is a workable approach.

7.2.3. Testing

1. In addition to running multi-threaded HDF5 tests regularly, it may also be useful to setup CI integration for testing HDF5 with one or both of the following tools:

[5] HDF5 PR #1659
[6] Assertions should be more debugger-oriented

- ThreadSanitizer - Tool from LLVM/Clang to detect data races. Could be useful with some tweaking, but it's currently in beta and known to have issues / produce false positives with the current approach to adapting HDF5 for multi-threaded usage.
- Helgrind - Valgrind tool for detecting several different types of errors in usage of POSIX pthreads. Has not yet been tested with multi-threaded HDF5. Generally only useful for POSIX pthreads, but can potentially be extended to other threading packages with its code annotation capabilities.

2. An interesting related project[7] explores forcing specific thread interleaving, which may be useful in very specific testing circumstances

[7] MultithreadedTC

Revision History

Version Number	Date	Comments
v1	Oct. 21, 2024	Version 1 drafted.

A. Appendix: 'Testframe' testing framework

The following sections give an overview of HDF5's 'testframe' testing framework which is contained within the `test/testframe.h` and `test/testframe.c` files. Details are only given for the types, functions, etc. that appear in the relevant header files.

A.1. Macros

```
#define MAXTESTNAME 64
```

The maximum number of bytes in a test name string, including the NUL terminator.

```
#define MAXTESTDESC 128
```

The maximum number of bytes in a test description string, including the NUL terminator.

```
/* Number of seconds to wait before killing a test (requires alarm(2)) */
#define H5_ALARM_SEC 1200 /* default is 20 minutes */
```

The default compile-time value for the `TestAlarmOn()` function.

```
/*
 * Predefined test verbosity levels.
 *
 * Convention:
 *
 * The higher the verbosity value, the more information printed.
 * So, output for higher verbosity also include output of all lower
 * verbosity.
 *
 *  Value     Description
 *  0         None:   No informational message.
 *  1                 "All tests passed"
 *  2                 Header of overall test
 *  3         Default: header and results of individual test
 *  4
 *  5         Low:    Major category of tests.
 *  6
 *  7         Medium: Minor category of tests such as functions called.
 *  8
 *  9         High:   Highest level.  All information.
 */
#define VERBO_NONE 0 /* None    */
#define VERBO_DEF  3 /* Default */
#define VERBO_LO   5 /* Low     */
#define VERBO_MED  7 /* Medium  */
#define VERBO_HI   9 /* High    */
```

Macros for the different test verbosity levels, for use with the `SetTestVerbosity()` function.

```
/*
 * Verbose queries
 * Only None needs an exact match.  The rest are at least as much.
 */
#define VERBOSE_NONE (HDGetTestVerbosity() == VERBO_NONE)
#define VERBOSE_DEF  (HDGetTestVerbosity() >= VERBO_DEF)
#define VERBOSE_LO   (HDGetTestVerbosity() >= VERBO_LO)
#define VERBOSE_MED  (HDGetTestVerbosity() >= VERBO_MED)
#define VERBOSE_HI   (HDGetTestVerbosity() >= VERBO_HI)
```

Macros for use with the `GetTestVerbosity()` function that return whether or not the current test verbosity level is at least set to the given level.

```
#define SKIPTEST  1 /* Skip this test */
#define ONLYTEST  2 /* Do only this test */
#define BEGINTEST 3 /* Skip all tests before this test */
```

Macros for values passed to the `SetTest()` function that determine the final behavior for a test that has been added to the list of tests.

```
#define MESSAGE(V, A)
do {
    if (TestFrameworkProcessID_g == 0 && TestVerbosity_g > (V))
        printf A;
} while(0)
```

Macro to print a message string `A` from the process with the process ID of 0, as long as the current test verbosity level setting is greater than `V`.

A.2. Enumerations

None defined

A.3. Structures

None defined

A.4. Global variables

```
int TestFrameworkProcessID_g = 0;
```

Contains an ID value for the current process which can be set during calls to `TestInit()`. Primarily used to control output from a parallel test so that it only comes from one MPI process, but also used for control flow.

```
int TestVerbosity_g = VERBO_DEF;
```

Contains the current setting for the level of output that tests should print. Defined values range from 0 - 9, with higher values implying more verbose output.

A.5. Functions

A.5.1. `TestInit()`

```
herr_t
TestInit(const char *ProgName, void (*TestPrivateUsage)(FILE *stream),
         int (*TestPrivateParser)(int argc, char *argv[]),
         herr_t (*TestSetupFunc)(void), herr_t (*TestCleanupFunc)(void),
         int TestProcessID);
```

Initializes test program-specific information and infrastructure. `TestInit()` should be called before any other function from this testing framework is called, but after other optional library setup functions, such as `H5open()` or `H5dont_atexit()`.

`ProgName` is simply a name for the test program (which may be different than the actual executable's name).

`TestPrivateUsage` is a pointer to a function that can print out additional test program usage help text that is specific to the test program. That function will be called by the `TestUsage()` function after it has printed out the more general test program command-line option information. May be `NULL`.

`TestPrivateParser` is a pointer to a function that will parse any command-line options that are specific to a test program. That function will be called by the `TestParseCmdLine()` function when a non-standard command-line option is found. May be `NULL`.

`TestSetupFunc` is a pointer to a function that can be used to setup any state needed before tests begin executing. If provided, this callback function will be called as part of `TestInit()` once the testing framework has been fully initialized. May be `NULL`.

`TestCleanupFunc` is a pointer to a function that can be used to clean up any state after tests have finished executing. If provided, this callback function will be called by `TestShutdown()` before the testing framework starts being shut down. May be `NULL`.

A.5.2. `TestShutdown()`

```
herr_t
TestShutdown(void);
```

Performs tear-down of the testing infrastructure by cleaning up internal state needed for running tests and freeing any associated memory. `TestShutdown()` should be called after any other function from this testing framework is called, and just before any optional library shutdown functions, such as `H5close()`.

A.5.3. `TestUsage()`

```
void
TestUsage(FILE *stream);
```

Prints out the usage help text for the test program to the given `FILE` pointer. Prints out a list of the general command-line options for the program, an optional list of additional test program-specific command-line options (if a `TestPrivateUsage` callback function was specified) and a list of all the tests and test descriptions for the test program. `stream` may be `NULL`, in which case `stdout` is used.

Note: when a parallel test calls `TestUsage()`, the output, including additional output from the optional callback specified in `TestInit()`, is only printed from the MPI process with rank value 0. Any collective operations should currently be avoided in the optional callback if one is provided.

A.5.4. `TestInfo()`

```
void
TestInfo(FILE *stream);
```

Prints out miscellaneous test information, which currently only includes the version of the HDF5 library that the test program was linked to, to the given `FILE` pointer. `stream` may be `NULL`, in which case `stdout` is used.

Note: when a parallel test calls `TestInfo()`, the output is only printed from the MPI process with rank value 0.

A.5.5. `AddTest()`

```
herr_t
AddTest(const char *TestName, void (*TestFunc)(const void *),
        void (*TestSetupFunc)(void *), void (*TestCleanupFunc)(void *),
        const void *TestData, size_t TestDataSize, const char *TestDescr);
```

This function adds a test to the list of tests that will be run when the `PerformTests()` function is called.

`TestName` is a short name string for the test and must be `MAXTESTNAME` bytes or less, including the NUL terminator. If the specified string begins with the character '-', the test will be set to be skipped by default.

`TestFunc` is a pointer to the function that will be called for the test. The function must return no value and accept a single `const void *` as an argument, which will point to any parameters to be passed to the test that are specified in `TestData`.

`TestSetupFunc` is an optional pointer to a function that will be called before the test's main test function is called. The function must return no value and accept a single `void *` as an argument, which will point to any parameters to be passed to the test that are specified in `TestData`.

`TestCleanupFunc` is an optional pointer to a function that will be called after the test's main test function has finished executing. The function must return no value and accept a single `void *` as an argument, which will point to any parameters to be passed to the test that are specified in `TestData`.

`TestData` is an optional pointer to test parameters that will be passed to the test's main test function when executed, as well as the test's optional setup and cleanup callbacks. If given, the testing framework will make a copy of the parameters according to the size specified in `TestDataSize`. If `TestData` is not `NULL`, `TestDataSize` must be a positive value. Otherwise, if `TestData` is `NULL`, `TestDataSize` must be 0.

`TestDataSize` is the size of the test parameter data to be passed to the test's main test function and setup and callback functions during execution. If `TestData` is not `NULL`, `TestDataSize` must be a positive value. Otherwise, if `TestData` is `NULL`, `TestDataSize` must be 0.

`TestDescr` is a short description string for the test and must be `MAXTESTDESC` bytes or less, including the NUL terminator. The string passed in `TestDescr` may be an empty string, but it is advised that test authors give a description to a test.

A.5.6. `TestParseCmdLine()`

```
herr_t
TestParseCmdLine(int argc, char *argv[]);
```

Parses the command-line options for the test program. Parse standard command-line options until it encounters a non-standard option, at which point it delegates to the `TestPrivateParser` callback function if one was specified by the `TestInit()` function. The following command-line options are the current standard options:

- `-verbose` / `-v` - Used to specify the level of verbosity of output from the test program. An extra argument must be provided to set the level, with the following being acceptable values: 'l' - low verbosity (value 5), 'm' - medium verbosity (value 7), 'h' - high verbosity (value 9), any number in 'int' range - set the verbosity to that value
- `-exclude` / `-x` - Used to specify a test that should be excluded when running. The short name of the test (as provided to `AddTest()`) must be provided as an extra argument to the command-line option.
- `-begin` / `-b` - Used to specify that a particular test should be the first test run from the set of tests. All tests before that test (in the order added by `AddTest()`) will be skipped and all tests after that test will be run as normal. The short name of the test (as provided to `AddTest()`) must be provided as an extra argument to the command-line option.
- `-only` / `-o` - Used to specify that only a particular test should be run. All other tests will be skipped. The short name of the test (as provided to `AddTest()`) must be provided as an extra argument to the command-line option.
- `-summary` / `-s` - Used to specify that a summary of all tests run should be printed out before the test program exits. No extra arguments should be provided.
- `-disable-error-stack` - Used to specify that printing out of HDF5 error stacks should be disabled. No extra arguments should be provided.
- `-help` / `-h` - Used to print out the usage help text of the test program.
- `-cleanoff` / `-c` - Used to specify that the `TestCleanupFunc()` callback function for each test should not clean up temporary HDF5 files when the test program exits. No extra arguments should be provided.
- `-maxthreads` / `-t` - Used to specify a maximum number of threads that a test program should be allowed to spawn in addition to the main thread. If a negative value is specified, test programs will be allowed to spawn any number of threads. The value 0 indicates that no additional threads should be spawned.

Note: `TestParseCmdLine()` requires that all standard command-line arguments must appear before any non-standard arguments that would be parsed by an optional argument parsing callback function specified in `TestInit()`.

Note: `TestParseCmdLine()` should not be called until all tests have been added by `AddTest()` since some of the command-line arguments that are parsed involve the ability to skip certain tests.

A.5.7. `PerformTests()`

```
herr_t
PerformTests(void);
```

Runs all tests added to the list of tests with `AddTest()` that aren't set to be skipped. For each test, the test's setup callback function (if supplied) will be called first, followed by the test's primary function and then the test's cleanup callback function (if supplied). Before each test begins, a timer is enabled by a call to `TestAlarmOn()` to prevent the test from running longer than desired. A call to `TestAlarmOff()` disables this timer after each test has finished.

A.5.8. `TestSummary()`

```
void
TestSummary(FILE *stream);
```

Used to print out a short summary after tests have run, which includes the name and description of each test and the number of errors that occurred while running that test. If a test was skipped, the number of errors for that test will show as "N/A". `stream` may be `NULL`, in which case `stdout` is used.

Note: when a parallel test calls `TestSummary()`, the output is only printed from the MPI process with rank value 0.

A.5.9. `GetTestVerbosity()`

```
int
GetTestVerbosity(void);
```

Returns the current test verbosity level setting.

A.5.10. `SetTestVerbosity()`

```
int
SetTestVerbosity(int newval);
```

Sets the current test verbosity level to the value specified by `newval`. If `newval` is negative, the test verbosity level is set to the lowest value (`VERBO_NONE`). If `newval` is greater than the highest verbosity value, it is set to the highest verbosity value (`VERBO_HI`). The function returns the previous test verbosity level setting.

A.5.11. `ParseTestVerbosity()`

```
herr_t
ParseTestVerbosity(char *argv);
```

Parses a string for a test verbosity level value, then sets the test verbosity level to that value. The string may be the character 'l' (for low verbosity), 'm' (for medium verbosity), 'h' (for high verbosity) or a number between 0-9, corresponding to the different predefined levels of test verbosity. If a negative number is specified, the test verbosity level is set to the default (`VERBO_DEF`). If a number greater than `VERBO_HI` is specified, the test verbosity level is set to `VERBO_HI`. If `ParseTestVerbosity()` can't parse the string, a negative value will be returned to indicate failure.

A.5.12. `GetTestExpress()`

```
int
GetTestExpress(void);
```

Returns the current `TestExpress` level setting from the 'h5test' framework. If the `TestExpress` level has not yet been set, it will be initialized to the default value (currently level 1, unless overridden at configuration time).

A.5.13. `SetTestExpress()`

```
void
SetTestExpress(int newval);
```

Sets the current `TestExpress` level setting to the value specified by `newval`. If `newval` is negative, the `TestExpress` level is set to the default value (currently level 1, unless overridden at configuration time). If `newval` is greater than the highest `TestExpress` level (3), it is set to the highest `TestExpress` level.

A.5.14. `GetTestSummary()`

```
bool
GetTestSummary(void);
```

Returns whether or not a test program should call `TestSummary()` to print out a summary of test results after tests have run.

A.5.15. `GetTestCleanup()`

```
int
GetTestCleanup(void);
```

Returns whether or not a test should clean up any temporary files it has created when it is finished running. If `true` is returned, the test should clean up temporary files. Otherwise, it should leave them in place. Each test

that has a cleanup callback should call `GetTestCleanup()` in that callback to determine whether or not to clean up temporary files.

A.5.16. `SetTestNoCleanup()`

```
void
SetTestNoCleanup(void);
```

Sets the temporary test file cleanup status to "don't cleanup temporary files", causing future calls to `GetTestCleanup()` to return `false` and inform tests that they should not clean up temporary test files they have created.

A.5.17. `GetTestNumErrs()`

```
int
GetTestNumErrs(void);
```

Returns the total number of errors recorded during the execution of the test program. This number is primarily used to determine whether the test program should exit with a success or failure value.

A.5.18. `IncTestNumErrs()`

```
void
IncTestNumErrs(void);
```

Simply increments the number of errors recorded for the test program.

A.5.19. `TestErrPrintf()`

```
int
TestErrPrintf(const char *format, ...);
```

Wrapper around `vfprintf` that includes a call to `IncTestNumErrs()`. The function returns the return value of `vfprintf`.

A.5.20. `SetTest()`

```
herr_t
SetTest(const char *testname, int action);
```

Given a test's short name and an action, modifies the behavior of how a test will run. The acceptable values for `action` are:

- `SKIPTEST` (1) - Skip this test.
- `ONLYTEST` (2) - Set this test to be the only test that runs. All other tests will be set to be skipped.

- `BEGINTEST` (3) - Set this test so that it will be the first test in the ordering that will run. All tests before it in the ordering will be set to be skipped and all tests after it will be untouched.

Other values for `action` will cause `SetTest()` to return a negative value for failure.

Multiple tests can be set to the value `ONLYTEST` in order to run a subset of tests. This is intended as a convenient alternative to needing to skip many other tests by setting them to the value `SKIPTEST`.

A.5.21. `GetTestMaxNumThreads()`

```
int
GetTestMaxNumThreads(void);
```

Returns the value for the maximum number of threads that a test program is allowed to spawn in addition to the main thread. This number is usually configured by a command-line argument passed to the test program and is intended for allowing tests to adjust their workload according to the resources of the testing environment.

The default value is -1, meaning that multi-threaded tests aren't limited in the number of threads they can spawn, but should still only use a reasonable amount of threads. The value 0 indicates that no additional threads should be spawned, which is primarily for testing purposes. The value returned by `GetTestMaxNumThreads()` is meaningless for non-multi-threaded tests.

A.5.22. `SetTestMaxNumThreads()`

```
herr_t
SetTestMaxNumThreads(int max_num_threads);
```

Sets the value for the maximum number of threads a test program is allowed to spawn in addition to the main thread for the test program. This number is usually configured by a command-line argument passed to the test program and is intended for allowing tests to adjust their workload according to the resources of the testing environment.

If `max_num_threads` is a negative value, test programs will be allowed to spawn any number of threads, though it is advised that test programs try to limit this to a reasonable number. The value 0 indicates that no additional threads should be spawned, which is primarily for testing purposes.

A.5.23. `TestAlarmOn()`

```
herr_t
TestAlarmOn(void);
```

Enables a timer for the test program using `alarm(2)`. If `alarm(2)` support is not available (the macro `H5_HAVE_ALARM` is not defined), the function does nothing. The default value passed to `alarm(2)` is defined by the macro `H5_ALARM_SEC` (currently 1200 seconds). If the `HDF5_ALARM_SECONDS` environment variable is defined, its value will be parsed by `strtoul()` and will override the default value.

A.5.24. `TestAlarmOff()`

```
void
TestAlarmOff(void);
```

Disables any previously set timer for a test program by calling `alarm(2)` with a value of 0. If `alarm(2)` support is not available (the macro `H5_HAVE_ALARM` is not defined), the function does nothing.

B. Appendix: 'H5test' testing framework

The following sections give an overview of HDF5's 'h5test' testing framework which is contained within the `test/h5test.h` and `test/h5test.c` files. Details are only given for the types, functions, etc. that appear in the relevant header files.

B.1. Macros

```
#define TESTING(WHAT)
```

Macro to print out the word "Testing" followed by the string specified in `WHAT`. As a side effect, the global variable `n_tests_run_g` is incremented. This macro does not print a newline and should therefore always be followed by a matching usage of one of the following macros.

```
#define PASSED()
#define H5_FAILED()
#define SKIPPED()
#define H5_WARNING()
#define TEST_ERROR
#define PUTS_ERROR(s)
#define STACK_ERROR
#define FAIL_STACK_ERROR
#define FAIL_PUTS_ERROR(s)
```

Macros to give information output about a test's pass/fail status and reasons for failure, if applicable.

`PASSED()` prints out " PASSED". As a side effect, the global variable `n_tests_passed_g` is incremented.

`H5_FAILED()` prints out "*FAILED*". As a side effect, the global variable `n_tests_failed_g` is incremented.

`SKIPPED()` prints out " -SKIP-". As a side effect, the global variable `n_tests_skipped_g` is incremented.

`H5_WARNING()` prints out "*WARNING*".

`TEST_ERROR` prints out "*FAILED*", as well as the file name, line number and function name where usage of the macro occurred and then uses `goto` to skip to the "error" label statement. As a side effect, the global variable `n_tests_failed_g` is incremented.

`PUTS_ERROR(s)` prints out the string specified in `s`, as well as the file name, line number and function name where usage of the macro occurred and then uses `goto` to skip to the "error" label statement.

`STACK_ERROR` prints out the contents of the default HDF5 error stack and then uses `goto` to skip to the "error" label statement.

`FAIL_STACK_ERROR` is similar to `STACK_ERROR`, but first prints out "*FAILED*", as well as the file name, line number and function name where usage of the macro occurred. As a side effect, the global variable `n_tests_failed_g` is incremented.

`FAIL_PUTS_ERROR(s)` is similar to `PUTS_ERROR(s)`, but first prints out "*FAILED*", as well as the file name, line number and function name where usage of the macro occurred. As a side effect, the global variable `n_tests_failed_g` is incremented.

```
#define AT()
```

Macro to print the file name, line number and function name where usage of the macro occurs.

```
#define TESTING_MULTIPART(WHAT)
#define TESTING_2(WHAT)
#define BEGIN_MULTIPART
#define END_MULTIPART
#define PART_BEGIN(part_name)
#define PART_END(part_name)
#define PART_ERROR(part_name)
#define PART_TEST_ERROR(part_name)
#define PART_EMPTY(part_name)
```

Counterpart to some of the above macros for tests which contain multiple sub-tests.

`TESTING_MULTIPART(WHAT)` prints out the word "Testing" followed by the string specified in `WHAT` and a newline.

The `TESTING_2()` macro is used to print out a slightly-indented "Testing" for each sub-test that is run. As a side effect, the global variable `n_tests_run_g` is incremented. This macro does not print a newline and should therefore always be followed by a usage of the `PASSED()` or `SKIPPED()` macros on success, or the `PART_ERROR()` or `PART_TEST_ERROR()` macros on failure.

`BEGIN_MULTIPART` and `END_MULTIPART` should surround all the sub-tests for a multi-part test. These macros keep track of whether any of the sub-tests failed and will use `goto` to skip to the "error" label statement when a failure does occur.

`PART_BEGIN(part_name)` and `PART_END(part_name)` should surround each individual sub-test in a multi-part test. These macros setup a label statement based on the name specified in `part_name`. When a failure occurs in a sub-test, the `PART_ERROR(part_name)` or `PART_TEST_ERROR(part_name)` macro will use `goto` to skip to the label statement that was setup, effectively moving on to the next sub-test instead of skipping the rest of the test.

`PART_ERROR(part_name)` uses `goto` to skip to the label statement with the name specified in the `part_name` parameter. As a side effect, the global variable `n_tests_failed_g` is incremented.

`PART_TEST_ERROR(part_name)` prints out "*FAILED*", as well as the file name, line number and function name where usage of the macro occurred and then uses `goto` to skip to the label statement with the name specified in `part_name`. As a side effect, the global variable `n_tests_failed_g` is incremented.

`PART_EMPTY(part_name)` uses `goto` to skip to the label statement with the name specified in the `part_name` parameter. This macro is mostly used for skipping an unimplemented sub-test in a multi-part test.

```
#define H5_FILEACCESS_VFD     0x01
#define H5_FILEACCESS_LIBVER  0x02
```

Macros for values that can be passed to the `h5_fileaccess_flags()` function.

```
#define H5_EXCLUDE_MULTIPART_DRIVERS     0x01
#define H5_EXCLUDE_NON_MULTIPART_DRIVERS 0x02
```

Macros for values that can be passed to the `h5_driver_uses_multiple_files()` function.

```
#define H5TEST_FILL_2D_HEAP_ARRAY(BUF, TYPE)
```

Utility macro to fill a 2-dimensional array `BUF` with increasing values starting from 0. `BUF` should point to a `struct { TYPE arr[...][...]; }`.

```
#define h5_free_const(mem) free((void *)(uintptr_t)mem)
```

Utility macro that can be used to cast away `const` from a pointer when freeing it in order to avoid compiler warnings.

B.2. Enumerations

None defined

B.3. Structures

None defined

B.4. Global variables

```
size_t n_tests_run_g;
size_t n_tests_passed_g;
size_t n_tests_failed_g;
size_t n_tests_skipped_g;
```

Global variables for tracking the number of tests run and how many of those tests passed, had failures or were skipped. Used for statistics and information printed at the end of test programs.

```
uint64_t vol_cap_flags_g;
```

Global variable to contain the capability flags for the current VOL connector in use, if applicable. Typically set at the start of a test program with a call to `H5Pget_vol_cap_flags()` and primarily used to skip tests which use functionality that a VOL connector doesn't advertise support for.

```
MPI_Info h5_io_info_g;
```

Global `MPI_Info` variable used by some parallel tests for passing MPI hints to parallel test operations.

B.5. Functions

B.5.1. `h5_test_init()`

```
void
h5_test_init(void);
```

Performs test framework initialization. Should be called by 'h5test' tests toward the beginning of the `main()` function after HDF5 has been initialized. Should **not** be called by 'testframe' tests.

B.5.2. `h5_restore_err()`

```
void
h5_restore_err(void);
```

Restores HDF5's default error handling function after its temporary replacement by `h5_test_init()`, which sets the error handling function to the `h5_errors()` callback function instead.

B.5.3. `h5_get_testexpress()`

```
int
h5_get_testexpress(void);
```

Returns the current `TestExpress` level setting, which determines whether a test program should expedite testing by skipping some tests. If the `TestExpress` level has not yet been set, it will be initialized to the default value (currently level 1, unless overridden at configuration time). The different `TestExpress` level settings have the following meanings:

- 0 - Tests should take as long as necessary
- 1 - Tests should take no more than 30 minutes
- 2 - Tests should take no more than 10 minutes
- 3 (or higher) - Tests should take no more than 1 minute

If the `TestExpress` level setting is not yet initialized, this function will first set a local variable to the value of the `H5_TEST_EXPRESS_LEVEL_DEFAULT` macro, if it has been defined. If the environment variable `HDF5TestExpress` is defined, its value will override the local variable's value. Acceptable values for the environment variable are the strings "0", "1" and "2"; any other string will cause the variable to be set to the value "3". Once the value for the local variable has been determined, `h5_get_testexpress()` returns that value.

B.5.4. `h5_set_testexpress()`

```
void
h5_set_testexpress(int new_val);
```

Sets the current `TestExpress` level setting to the value specified by `new_val`. If `new_val` is negative, the `TestExpress` level is set to the default value (currently level 1, unless overridden at configuration time). If `new_val` is greater than the highest `TestExpress` level (3), it is set to the highest `TestExpress` level.

B.5.5. `h5_fileaccess()`

```
hid_t
h5_fileaccess(void);
```

Wrapper function around `h5_get_vfd_fapl()` and `h5_get_libver_fapl()` which returns a File Access Property List that has potentially been configured with a non-default file driver and library version bounds

setting. Should generally be the primary way for tests to obtain a File Access Property List to use when testing with different VFDs would not be problematic.

B.5.6. `h5_fileaccess_flags()`

```
hid_t
h5_fileaccess_flags(unsigned flags);
```

Counterpart to `h5_fileaccess()` which allows the caller to specify which parts of the File Access Property List should be modified after it is created. `flags` may be specified as one of the macro values `H5_FILEACCESS_VFD` or `H5_FILEACCESS_LIBVER`, where the former specifies that the file driver of the FAPL should be modified, while the latter specifies that the library version bounds setting of the FAPL should be modified. `flags` can also be specified as a bit-wise OR of the two values, in which case behavior is equivalent to `h5_fileaccess()`.

B.5.7. `h5_get_vfd_fapl()`

```
herr_t
h5_get_vfd_fapl(hid_t fapl_id);
```

Modifies the File Access Property List specified in `fapl_id` by setting a new Virtual File Driver on it with default configuration values. The Virtual File Driver to be used is chosen according to the value set for either the `HDF5_DRIVER` or `HDF5_TEST_DRIVER` environment variable, with preference given to `HDF5_DRIVER`. These environment variables may be set to one of the following strings:

- ”sec2” - Default sec2 driver; no configuration supplied
- ”stdio” - C STDIO driver; no configuration supplied
- ”core” - In-memory driver; 1MiB increment size and backing store enabled
- ”core_paged” - In-memory driver; 1MiB increment size and backing store enabled; write tracking enabled and 4KiB page size
- ”split” - Multi driver with the file’s metadata being placed into a file with a ”.meta” suffix and the file’s raw data being placed into a file with a ”.raw” suffix
- ”multi” - Multi driver with the file data being placed into several different files with the suffixes ”m.h5” (metadata), ”s.h5” (superblock, userblock and driver info data), ”b.h5” (b-tree data), ”r.h5” (dataset raw data), ”g.h5” (global heap data), ”l.h5” (local heap data) and ”o.h5” (object header data)
- ”family” - Family driver with a family file size of 100MiB and with a default File Access Property List used for accessing the family file members. A different family file size can be specified in the environment variable as an integer value of bytes separated from the string ”family” with whitespace, e.g. ”family 52428800” would specify a family file size of 50MiB.
- ”log” - Log driver using `stderr` for output, with the `H5FD_LOG_LOC_IO` and `H5FD_LOG_ALLOC` flags set and 0 for the buffer size. A different set of flags may be specified for the driver as an integer value (corresponding to a bit-wise OR of flags) separated from the string ”log” with whitespace, e.g. ”log 14” would equate to the flag `H5FD_LOG_LOC_IO`.

- ”direct” - Direct driver with a 4KiB block size, a 32KiB copy buffer size and a 1KiB memory alignment setting. If the direct driver is not enabled when HDF5 is built, `h5_get_vfd_fapl()` will return an error.
- ”splitter” - Splitter driver using the default (sec2) VFD for both the read/write and write-only channels, an empty log file path and set to not ignore errors on the write-only channel
- ”onion” - support not currently implemented; will cause `h5_get_vfd_fapl()` to return an error.
- ”subfiling” - Subfiling VFD with a default configuration of 1 I/O concentrator per node, a 32MiB stripe size and using the IOC VFD with 4 worker threads. `MPI_COMM_WORLD` and `MPI_INFO_NULL` are used for the MPI parameters.
- ”mpio” - MPI I/O VFD with `MPI_COMM_WORLD` and `MPI_INFO_NULL` used for the MPI parameters.
- ”mirror” - support not currently implemented; will cause `h5_get_vfd_fapl()` to return an error.
- ”hdfs” - support not currently implemented; will cause `h5_get_vfd_fapl()` to return an error.
- ”ros3” - support not currently implemented; will cause `h5_get_vfd_fapl()` to return an error.

Other values for the environment variables will cause `h5_get_vfd_fapl()` to return an error.

Returns a non-negative value on success and a negative value on failure.

B.5.8. `h5_get_libver_fapl()`

```
herr_t
h5_get_libver_fapl(hid_t fapl_id);
```

Modifies the File Access Property List specified in `fapl_id` by setting library version bound values on it according to the value set for the `HDF5_LIBVER_BOUNDS` environment variable. Currently, the only valid value for this environment variable is the string ”latest”, which will cause this function to set the low and high version bounds both to `H5F_LIBVER_LATEST`. Other values for the environment variable will cause this function to fail and return a negative value.

B.5.9. `h5_cleanup()`

```
int
h5_cleanup(const char *base_name[], hid_t fapl);
```

Used to cleanup temporary files created by a test program. `base_name` is an array of filenames without suffixes to clean up. The last entry in the array must be `NULL`. For each filename specified, `h5_cleanup()` will generate a VFD-dependent filename with `h5_fixname()` according to the given File Access Property List `fapl`, then call `H5Fdelete()` on the resulting filename. `fapl` will be closed after all files are deleted. If the environment variable `HDF5_NOCLEANUP` has been defined, this function will have no effect and `fapl` will be left open. Returns 1 if cleanup was performed and 0 otherwise.

B.5.10. h5_delete_all_test_files()

```
void
h5_delete_all_test_files(const char *base_name[], hid_t fapl);
```

Used to cleanup temporary files created by a test program. `base_name` is an array of filenames without suffixes to clean up. The last entry in the array must be `NULL`. For each filename specified, this function will generate a VFD-dependent filename with `h5_fixname()` according to the given File Access Property List `fapl`, then call `H5Fdelete()` on the resulting filename. `fapl` will **not** be closed after all files are deleted. `h5_delete_all_test_files()` always performs file cleanup, regardless of if the `HDF5_NOCLEANUP` environment variable has been defined.

B.5.11. h5_delete_test_file()

```
void
h5_delete_test_file(const char *base_name, hid_t fapl);
```

Used to cleanup a temporary file created by a test program. `base_name` is a filename without a suffix to clean up. This function will generate a VFD-dependent filename with `h5_fixname()` according to the given File Access Property List `fapl`, then call `H5Fdelete()` on the resulting filename. `fapl` will **not** be closed after the file is deleted. `h5_delete_test_file()` always performs file cleanup, regardless of if the `HDF5_NOCLEANUP` environment variable has been defined.

B.5.12. h5_fixname()

```
char
*h5_fixname(const char *base_name, hid_t fapl, char *fullname,
            size_t size);
```

Given a base filename without a suffix, `base_name`, and a File Access Property List, `fapl`, generates a filename according to the configuration set on `fapl`. The resulting filename is copied to `fullname`, which is `size` bytes in size, including space for the `NUL` terminator.

`h5_fixname()` is the primary way that tests should create filenames, as it accounts for the possibility of a test being run with a non-default Virtual File Driver that may require a specialized filename (e.g., the family driver). It also allows tests to easily output test files to a different directory by setting the `HDF5_PREFIX` (for serial tests) or `HDF5_PARAPREFIX` (for parallel tests) environment variable.

Returns the `fullname` parameter on success, or `NULL` on failure.

B.5.13. h5_fixname_superblock()

```
char
*h5_fixname_superblock(const char *base_name, hid_t fapl, char *fullname,
                       size_t size);
```

`h5_fixname_superblock()` is similar to `h5_fixname()`, but generates the filename that would need to be opened to find the logical HDF5 file's superblock. Useful for when a file is to be opened with `open(2)` but the `h5_fixname()` string contains stuff like format strings.

B.5.14. `h5_fixname_no_suffix()`

```
char
*h5_fixname_no_suffix(const char *base_name, hid_t fapl, char *fullname,
                      size_t size);
```

`h5_fixname_no_suffix()` is similar to `h5_fixname()`, but generates a filename that has no suffix, where the filename from `h5_fixname()` would typically have ".h5".

B.5.15. `h5_fixname_printf()`

```
char
*h5_fixname_printf(const char *base_name, hid_t fapl, char *fullname,
                   size_t size);
```

`h5_fixname_printf()` is similar to `h5_fixname()`, but generates a filename that can be passed through a `printf`-style function to obtain the final, processed filename. Essentially replaces all % characters that would be used by a file driver with %%.

B.5.16. `h5_no_hwconv()`

```
void
h5_no_hwconv(void);
```

Temporarily turns off hardware datatype conversions in HDF5 during testing by calling `H5Tunregister()` to unregister all the hard conversion pathways. Useful to verify that datatype conversions for different datatypes still work correctly when emulated by the library.

B.5.17. `h5_rmprefix()`

```
const char
*h5_rmprefix(const char *filename);
```

"Removes" a prefix from a filename by searching for the first occurence of ":" and returning a pointer into the filename just past that occurrence. No actual changes are made to the file name.

B.5.18. `h5_show_hostname()`

```
void
h5_show_hostname(void);
```

Prints out `hostname`-like information. Also prints out each MPI process' rank value if HDF5 is built with parallel functionality enabled and MPI is initialized. Otherwise, if HDF5 is built with thread-safe functionality enabled and MPI is not initialized or HDF5 is not built with parallel functionality enabled, also prints out thread ID values.

B.5.19. `h5_get_file_size()`

```
h5_stat_size_t
h5_get_file_size(const char *filename, hid_t fapl);
```

Returns the size in bytes of the file with the given filename `filename`. A File Access Property List specified for `fapl` will modify how the file size is calculated. If `H5P_DEFAULT` is passed for `fapl`, `stat` or the platform equivalent is used to determine the file size. Otherwise, the calculation depends on the file driver set on `fapl`. For example, a FAPL setup with the MPI I/O driver will cause `h5_get_file_size()` to use `MPI_File_get_size()`, while a FAPL setup with the family driver will cause `h5_get_file_size()` to sum the sizes of the files in the family file.

B.5.20. `h5_make_local_copy()`

```
int
h5_make_local_copy(const char *origfilename, const char *local_copy_name);
```

Given a file with the filename `origfilename`, makes a byte-for-byte copy of the file, which is then named `local_copy_name`, using POSIX I/O. Returns 0 on success and a negative value on failure. This function is useful for making copies of test files that are under version control. Tests should make a copy of the original file and then operate on the copy.

B.5.21. `h5_duplicate_file_by_bytes()`

```
int
h5_duplicate_file_by_bytes(const char *orig, const char *dest);
```

Similar to `h5_make_local_copy()`, but uses C stdio functions. Returns 0 on success and a negative value on failure.

B.5.22. `h5_compare_file_bytes()`

```
int
h5_compare_file_bytes(char *fname1, char *fname2);
```

Performs a byte-for-byte comparison of two files with the names `fname1` and `fname2`. Returns 0 if the files are identical and -1 otherwise.

B.5.23. h5_verify_cached_stabs()

```
herr_t
h5_verify_cached_stabs(const char *base_name[], hid_t fapl);
```

Verifies that all groups in a set of files have their symbol table information cached, if present and if their parent group also uses a symbol table. `base_name` is an array of filenames without suffixes, where the last entry must be `NULL`. `fapl` is the File Access Property List used to open each of the files in `base_name`. Returns a non-negative value on success and a negative value on failure.

B.5.24. h5_get_dummy_vfd_class()

```
H5FD_class_t
*h5_get_dummy_vfd_class(void);
```

Allocates and returns a pointer to a "dummy" Virtual File Driver class which is generally non-functional. Must be freed by the caller with `free()` once it is no longer needed.

B.5.25. h5_get_dummy_vol_class()

```
H5VL_class_t
*h5_get_dummy_vol_class(void);
```

Allocates and returns a pointer to a "dummy" Virtual Object Layer connector class which is generally non-functional. Must be freed by the caller with `free()` once it is no longer needed.

B.5.26. h5_get_version_string()

```
const char
*h5_get_version_string(H5F_libver_t libver);
```

Given a particular library version bound value, `libver`, translates the value into a canonical string value that is returned.

B.5.27. h5_check_if_file_locking_enabled()

```
herr_t
h5_check_if_file_locking_enabled(bool *are_enabled);
```

Checks if file locking is enabled on the system by creating a temporary file and calling `flock()` or the platform equivalent on it. A non-negative value is return on success and a negative value is returned otherwise. If this function succeeds and `are_enabled` is set to `true`, file locking is enabled on the system. Otherwise, it should be assumed the file locking is not enabled or is problematic.

B.5.28. h5_check_file_locking_env_var()

```
void
h5_check_file_locking_env_var(htri_t *use_locks,
                              htri_t *ignore_disabled_locks);
```

Parses the value of the HDF5_USE_FILE_LOCKING environment variable, if set, and returns whether or not file locking should be used and whether or not failures should be ignored when attempting to use file locking on a system where it is disabled.

B.5.29. h5_using_native_vol()

```
herr_t
h5_using_native_vol(hid_t fapl_id, hid_t obj_id, bool *is_native_vol);
```

Checks if the VOL connector being used for testing is the library's native VOL connector. One of either fapl_id or obj_id must be provided as a reference point to be checked; if both are provided, checking of obj_id takes precedence. H5I_INVALID_HID should be specified for the parameter that is not provided.

obj_id must be the ID of an HDF5 object that is accessed with the VOL connector to check. If obj_id is provided, the entire VOL connector stack is checked to see if it resolves to the native VOL connector. If only fapl_id is provided, only the top-most VOL connector set on fapl_id is checked against the native VOL connector.

A non-negative value is return on success and a negative value is returned otherwise.

B.5.30. h5_get_test_driver_name()

```
const char
*h5_get_test_driver_name(void);
```

Returns a pointer to the name of the VFD being used for testing. If the environment variable HDF5_DRIVER or HDF5_TEST_DRIVER has been set, the value set for that variable is returned, with preference given to the HDF5_DRIVER environment variable if both are set. Otherwise, the name of the library's default VFD is returned.

B.5.31. h5_using_default_driver()

```
bool
h5_using_default_driver(const char *drv_name);
```

Returns true if the name of the VFD being used for testing matches the name of the library's default VFD and false otherwise. If drv_name is NULL, h5_get_test_driver_name() is called to obtain the name of the VFD in use before making the comparison.

B.5.32. h5_using_parallel_driver()

```
herr_t
h5_using_parallel_driver(hid_t fapl_id, bool *driver_is_parallel);
```

Checks if the VFD set on `fapl_id` is a parallel-enabled VFD that supports MPI. A VFD must have set the `H5FD_FEAT_HAS_MPI` feature flag to be considered as a parallel-enabled VFD. `fapl_id` may be `H5P_DEFAULT`. A non-negative value is return on success and a negative value is returned otherwise.

B.5.33. h5_driver_is_default_vfd_compatible()

```
herr_t
h5_driver_is_default_vfd_compatible(hid_t fapl_id,
                                    bool *default_vfd_compatible);
```

Checks if the VFD set on `fapl_id` creates files that are compatible with the library's default VFD. For example, the core and MPI I/O drivers create files that are compatible with the library's default VFD, while the multi and family drivers do not since they split the HDF5 file into several different files. This check is helpful for skipping tests that use pre-generated testing files. VFDs that create files which aren't compatible with the default VFD will generally not be able to open these pre-generated files and those particular tests will fail.

`fapl_id` may be `H5P_DEFAULT`. A non-negative value is return on success and a negative value is returned otherwise.

B.5.34. h5_driver_uses_multiple_files()

```
bool
h5_driver_uses_multiple_files(const char *drv_name, unsigned flags);
```

Returns `true` if the given VFD name, `drv_name`, matches the name of a VFD which stores data using multiple files, according to the specified `flags` and `false` otherwise. If `drv_name` is `NULL`, the `h5_get_test_driver_name()` function is called to obtain the name of the VFD in use before making the comparison. The values for `flags` are as follows:

- `H5_EXCLUDE_MULTIPART_DRIVERS` - This flag excludes any drivers which store data using multiple files which, together, make up a single logical file. These are drivers like the split, multi and family drivers.
- `H5_EXCLUDE_NON_MULTIPART_DRIVERS` - This flag excludes any drivers which store data using multiple files which are separate logical files. The splitter driver is an example of this type of driver.

B.5.35. h5_local_rand()

```
int
h5_local_rand(void);
```

Function to return a random number without modifying state for the `rand()`/`random()` functions.

B.5.36. h5_local_srand()

```
void
h5_local_srand(unsigned int seed);
```

Function to seed the `h5_local_rand()` function without modifying state for the `rand()`/`random()` functions.

B.5.37. h5_szip_can_encode()

```
int
h5_szip_can_encode(void);
```

Returns a value that indicates whether or not the library's SZIP filter has encoding/decoding enabled. Returns 1 if encoding and decoding are enabled. Returns 0 if only decoding is enabled. Otherwise, returns -1.

B.5.38. h5_set_info_object()

```
int
h5_set_info_object(void);
```

Utility function for parallel HDF5 tests which parses the `HDF5_MPI_INFO` environment variable for ";"-delimited key=value pairs and sets them on the `h5_io_info_g` MPI Info global variable for later use by testing. Returns 0 on success and a negative value otherwise.

B.5.39. h5_dump_info_object()

```
void
h5_dump_info_object(MPI_Info info);
```

Given an MPI Info object, `info`, iterates through all the keys set on the Info object and prints them out as key=value pairs.

B.5.40. getenv_all()

```
char
*getenv_all(MPI_Comm comm, int root, const char *name);
```

Retrieves the value of the environment variable `name`, if set, on the MPI process with rank value `root` on the MPI Communicator `comm`, then broadcasts the result to other MPI processes in `comm`. Collective across the MPI Communicator specified in `comm`. If MPI is not initialized, simply calls `getenv(name)` and returns a pointer to the result. `NULL` is returned if the environment variable `name` is not set.

Note: the pointer returned by this function is only valid until the next call to `getenv_all()` and the data stored there must be copied somewhere else before any further calls to `getenv_all()` take place.

B.5.41. `h5_send_message()`

```
void
h5_send_message(const char *file, const char *arg1, const char *arg2);
```

Utility function to facilitate inter-process communication by "sending" a message with a temporary file. `file` is the name of the temporary file to be created. `arg1` and `arg2` are strings to be written to the first and second lines of the file, respectively, and may both be `NULL`.

B.5.42. `h5_wait_message()`

```
herr_t
h5_wait_message(const char *file);
```

Utility function to facilitate inter-process communication by waiting until a temporary file written by the `h5_send_message()` function is available for reading. `file` is the name of the file being waited on and should match the filename provided to `h5_send_message()`. This function repeatedly attempts to open a file with the given filename until it is either successful or times out. The temporary file is removed once it has been successfully opened. A non-negative value is return on success and a negative value is returned otherwise.

C. Appendix: 'Testpar' testing framework

The following sections give an overview of HDF5's 'testpar' testing framework which is contained within the `testpar/testpar.h` and `testpar/testpar.c` files. Details are only given for the types, functions, etc. that appear in the relevant header files.

C.1. Macros

```
#define MAINPROCESS (!mpi_rank)
```

Convenience macro that returns a `true` value for the MPI process with rank value 0, and a `false` value for MPI processes with rank values that aren't 0. Used to separate test logic that should only be executed on the MPI process with rank value 0 from test logic that should be executed on other MPI processes.

```
#define MESG(mesg)
```

Macro to print out a message string `mesg` as long as the test verbosity level setting is at least medium (7) and the string isn't empty.

Note: The specified message string will be printed by any MPI process which uses this macro. Test programs should avoid using this macro on all MPI processes to prevent large amounts of output.

```
#define MPI_BANNER(mesg)
```

Macro to print out a message string `mesg` surrounded by a block of '-' characters. If the test verbosity level setting is at least medium (7), the message string will be printed out by all MPI processes that use the macro. Otherwise, the message string will only be printed out by the MPI process with rank value 0.

```
#define VRFY_IMPL(val, mesg, rankvar)
#define VRFY_G(val, mesg) VRFY_IMPL(val, mesg, mpi_rank_g)
#define VRFY(val, mesg)   VRFY_IMPL(val, mesg, mpi_rank)
#define INFO(val, mesg)
```

Set of macros used for verifying that the condition or variable specified in `val` is `true`.

If the condition or variable specified in `val` is `true`, the message string specified in `mesg` will be printed using the `MESG()` macro (as long as the test verbosity level setting is at least medium (7)). Otherwise, an error message will be printed out. When the condition or variable specified in `val` is `false`, the `VRFY` macros will call `MPI_Abort()` to terminate the test program, as long as the test verbosity level setting is below medium (7).

The third parameter to the `VRFY_IMPL()` macro should be the name of the variable that contains the MPI process rank value. `VRFY()` and `VRFY_G()` are simply convenience wrappers for when that variable is named `mpi_rank` or `mpi_rank_g`, respectively.

```
#define SYNC(comm)
```

Macro to perform an `MPI_Barrier()` operation on the specified MPI Communicator. Informational before and after messages will be printed using the `MPI_BANNER()` macro.

```
#define H5_PARALLEL_TEST
```

This macro is defined primarily so that other headers included by either `testpar.h` or the test program itself can determine whether they're being included from a parallel test and can adjust functionality accordingly.

```
#define FACC_DEFAULT 0x0 /* default */
#define FACC_MPIO    0x1 /* MPIO */
#define FACC_SPLIT   0x2 /* Split File */
```

These macros are definitions for the `create_faccess_plist()` which are used to specify modifications to be made to the File Access Property List that is returned from the function.

```
#define DXFER_COLLECTIVE_IO  0x1 /* Collective IO */
#define DXFER_INDEPENDENT_IO 0x2 /* Independent IO collectively */
```

These macros are used by various parallel tests to control test behavior depending on whether the test is operating in collective or independent I/O mode.

```
#define BYROW 1 /* divide into slabs of rows */
#define BYCOL 2 /* divide into blocks of columns */
#define ZROW  3 /* same as BYCOL except process 0 gets 0 rows */
#define ZCOL  4 /* same as BYCOL except process 0 gets 0 columns */
```

These macros are used by various parallel tests to determine how to split a hyperslab selection among the available MPI processes when performing parallel dataset I/O.

```
#define IN_ORDER     1
#define OUT_OF_ORDER 2
```

These macros are used by various parallel tests to determine the method for selecting points when using a point selection for parallel dataset I/O.

```
#define MAX_ERR_REPORT 10 /* Maximum number of errors reported */
```

This macro is intended to put a limit on the number of errors that a parallel test program should report in cases such as data verification loops, where several errors could be reported at once from multiple processes.

C.2. Enumerations

```
enum H5TEST_COLL_CHUNK_API {
    API_NONE = 0,
    API_LINK_HARD,
    API_MULTI_HARD,
    API_LINK_TRUE,
    API_LINK_FALSE,
    API_MULTI_COLL,
    API_MULTI_IND
};
```

Enumeration value which some parallel tests use to determine how to perform parallel I/O

```
typedef enum {
    IND_CONTIG,  /* Independent IO on contiguous datasets */
```

```
    COL_CONTIG,  /* Collective IO on contiguous datasets */
    IND_CHUNKED, /* Independent IO on chunked datasets */
    COL_CHUNKED  /* Collective IO on chunked datasets */
} ShapeSameTestMethods;
```

Enumeration value which some parallel tests use to determine how to perform parallel I/O and on what type of datasets.

C.3. Structures

None defined

C.4. Global variables

None defined

C.5. Functions

C.5.1. `create_faccess_plist()`

```
hid_t
create_faccess_plist(MPI_Comm comm, MPI_Info info, int l_facc_type);
```

Utility function to create a File Access Property List. If the macro value `FACC_DEFAULT` is passed for `l_facc_type`, the FAPL returned will be a default FAPL for serial file access. If the value `FACC_MPIO` is passed for `l_facc_type`, the FAPL will be setup for accessing a file through the MPI I/O file driver according to the MPI Communicator and Info objects specified in `comm` and `info`. Collective metadata reads and writes will also be enabled on the FAPL. If the value `FACC_MPIO | FACC_SPLIT` is passed for `l_facc_type`, the FAPL will be setup for accessing a file through the multi file driver, with the file's metadata and raw data being split into separate files. The multi file driver will use the MPI I/O driver for managing the file data according to the MPI Communicator and Info objects specified in `comm` and `info`. Collective metadata reads and writes will **not** be enabled on the FAPL.

C.5.2. `point_set()`

```
void
point_set(hsize_t start[], hsize_t count[], hsize_t stride[],
          hsize_t block[], size_t num_points, hsize_t coords[],
          int order);
```

Utility function to translate a hyperslab selection, specified by the `start`, `count`, `stride` and `block` parameters, into a point selection by populating the `coords` array. `order` may be specified as either of the macro values `IN_ORDER` or `OUT_OF_ORDER`. `IN_ORDER` causes the `coords` array to be populated in ascending order while iterating through the hyperslab selection's elements; `OUT_OF_ORDER` causes the `coords` array to be populated in descending order, resulting in the last selected element appearing first in the

`coords` array. If `order` is specified as `OUT_OF_ORDER`, the `num_points` parameter specifies the number of elements to translate from the hyperslab selection and should be less than or equal to the number of points that the `coords` array has allocated space for.

D. Appendix: Examples

Error checking is omitted in the following examples for simplicity.

D.1. Example skeleton for a 'testframe' HDF5 test

```c
#include "testframe.h"

int
main(int argc, char **argv)
{
    H5open();

    /* Initialize testing framework */
    TestInit(argv[0], NULL, NULL, NULL, NULL, 0);

    /* Add each test */
    AddTest("test1", test1(), test1_setup(), test1_cleanup(),
            &test1_data, sizeof(test1_data), "Runs test 1");
    AddTest("test2", test2(), test2_setup(), test2_cleanup(),
            &test2_data, sizeof(test2_data), "Runs test 2");

    /* Parse command line arguments */
    TestParseCmdLine(argc, argv);

    /* Display testing information */
    TestInfo(stdout);

    /* Start test timer */
    TestAlarmOn();

    /* Run tests */
    PerformTests();

    /* Display test summary, if requested */
    if (GetTestSummary())
        TestSummary(stdout);

    /* Turn test timer off */
    TestAlarmOff();

    num_errs = GetTestNumErrs();

    /* Release test infrastructure */
    TestShutdown();
```

```
    if (num_errs > 0)
        exit(EXIT_FAILURE);
    else
        exit(EXIT_SUCCESS);
}
```

D.2. Example skeleton for a parallel 'testframe' HDF5 test

```
#include "testframe.h"

int
main(int argc, char **argv)
{
    int mpi_size; /* MPI communicator size */
    int mpi_rank; /* MPI process rank value */

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    H5open();

    /* Initialize testing framework; pass MPI rank value
       for process ID */
    TestInit(argv[0], NULL, NULL, NULL, NULL, mpi_rank);

    /* Add each test */
    AddTest("test1", test1(), test1_setup(), test1_cleanup(),
            &test1_data, sizeof(test1_data), "Runs test 1");
    AddTest("test2", test2(), test2_setup(), test2_cleanup(),
            &test2_data, sizeof(test2_data), "Runs test 2");

    /* Parse command line arguments */
    TestParseCmdLine(argc, argv);

    /* Display testing information */
    TestInfo(stdout);

    /* Start test timer */
    TestAlarmOn();

    /* Run tests */
    PerformTests();
```

```
    /* Display test summary, if requested */
    if (GetTestSummary())
        TestSummary(stdout);

    /* Turn test timer off */
    TestAlarmOff();

    /* Gather errors from all processes */
    num_errs = GetTestNumErrs();
    MPI_Allreduce(MPI_IN_PLACE, &num_errs, 1, MPI_INT,
                  MPI_MAX, MPI_COMM_WORLD);

    /* Release test infrastructure */
    TestShutdown();

    MPI_Finalize();

    if (num_errs > 0)
        exit(EXIT_FAILURE);
    else
        exit(EXIT_SUCCESS);
}
```

D.3. Example skeleton for a multi-threaded HDF5 test

```
#include "testframe.h"

static void
test1(const void *params)
{
    unsigned max_num_threads = GetTestMaxNumThreads();
    unsigned timeout = *(unsigned *)params;

    /* Calculate amount of work to do */
    ...

    /* Do work */
    ...

    return;
}

int
main(int argc, char **argv)
```

```
{
    unsigned runtime;
    unsigned num_subtests;
    unsigned subtest_timeout;
    int      TestExpress;
    int      num_errs;

    /* Initialize testing framework */
    TestInit(argv[0], NULL, NULL, NULL, NULL, 0);

    TestExpress = GetTestExpress();
    if (TestExpress == 0) {
        runtime = 0; /* Run with no timeout (runtime = 0) or pick a
                        reasonable timeout */
    }
    else if (TestExpress == 1) {
        runtime = 1800; /* 30 minute timeout */
    }
    else if (TestExpress == 2) {
        runtime = 600; /* 10 minute timeout */
    }
    else {
        runtime = 60; /* 1 minute timeout */
    }

    /* Calculate timeout for each sub-test */
    subtest_timeout = (runtime - MARGIN) / num_subtests;

    /* Add each test, passing the timeout as a parameter */
    AddTest("test1", test1(), NULL, test1_cleanup(),
            &subtest_timeout, sizeof(unsigned), "Runs test 1");
    AddTest("test2", test2(), NULL, test2_cleanup(),
            &subtest_timeout, sizeof(unsigned), "Runs test 2");

    /* Parse command line arguments, including the maximum number of
       threads to use */
    TestParseCmdLine(argc, argv);

    /* Display testing information */
    TestInfo(stdout);

    /* Start test timer; assumes refactoring of TestAlarmOn to accept
       a specified timeout. TestAlarmOn could also be refactored to
       handle this itself, given its own information about the TestExpress
       value */
    TestAlarmOn(runtime);
```

```
    /* Run sub-tests, with the test framework being responsible for spawning
       threads to run each test function. If performance becomes an issue,
       threads can be spawned once and the entire test can be encapsulated
       in a single function. Also, the testing framework may need
       infrastructure to check against the maximum allowed runtime after
       each sub-test in case a particular sub-test spills over its time
       budget. */
    PerformTests();

    /* Display test summary, if requested */
    if (GetTestSummary())
        TestSummary(stdout);

    /* Not strictly necessary to turn the alarm off, but the test alarm
       should generally be left on until right before exiting since even
       the cleanup step may be prone to hanging */
    TestAlarmOff();

    num_errs = GetTestNumErrs();

    /* Release test infrastructure */
    TestShutdown();

    if (num_errs > 0)
        exit(EXIT_FAILURE);
    else
        exit(EXIT_SUCCESS);
}
```