HDF Software Process

Lessons Learned or Success Factors

Mike Folk and Elena Pourmal
NCSA HDF Group

December 20, 2004

I. Introduction

This is a collection of thoughts on the HDF group approach to software development, and on software engineering and technical, business and social group practices as integral, mutually bound parts of the software development process.

The HDF project that is briefly described in the next section has been a very successful software project for the past 15 years. We would like to share our software development experience and practices, the factors that have lead us to these achievements, with our users, and with software developers who work within the walls of universities or in not-for-profit open source software groups. These factors include the following:

- Strong, responsible, and continuing relationships with users
- An approach to needs identification, software design, and software implementation based on sound principles of software engineering
- Effective technical processes for developing, testing, integrating and maintaining software
- Business and social processes based on sound group management principles

These factors are little more than platitudes, however. The manner in which they are successfully applied can only be understood by examining the details. In these pages, we describe some of the details, emphasizing mostly those areas in which we have had success.

II. What is HDF?

To understand the software engineering practices of the HDF project, it is useful to know a bit about HDF itself. In this section we describe HDF, and some aspects of the HDF implementation that make the project challenging. We focus on HDF5, but HDF4 has similar characteristics. For more details, visit http://hdf.ncsa.uiuc.edu.

Briefly, HDF (Hierarchical Data Format) is a project at the National Center for Supercomputing Applications) that has developed two file formats (HDF4 and HDF5), together with I/O libraries and other software for storing, managing and archiving large complex scientific and other data. HDF software has a University of Illinois License, but is open source and free for any use. Because it is supported by a number of organizations for which it is a critical technology, the format is well supported and maintained. Detailed information about HDF, its users and applications, can be found at http://hdf.ncsa.uiuc.edu.

Both HDF4 and HDF5 were designed to be a general scientific format, adaptable to virtually any scientific or engineering application, and also have been used successfully in non-technical areas. HDF5 is particularly good at dealing with data where complexity and scalability are important. Data of virtually any type or size can be stored in HDF5, including complex data structures and data types. HDF5 is portable, running on most operating systems and machines. HDF5 is scalable – it works well in high end computing environments, and can accommodate data objects of almost any size or multiplicity. It is also efficient, providing fast access to data, including parallel I/O. It also can store large amounts of data efficiently – it has built-in compression, or applications can also provide their own special-purpose compression.

HDF4 and HDF5 are both widely used in industry, academia, and government. There are more then 200 distinct applications of the formats, and an estimated 1.6 million users of NASA data alone. It is also the base format for a number of community standards, such as HDF-EOS, the standard for NASA's enormous Earth Observing System (http://hdfeos.gsfc.nasa.gov/hdfeos/index.cfm) , and NeXus, the standard for Neutron, X-ray and Muon Science (http://www.nexus.anl.gov/index.html) . It is useful to think about HDF software in terms of layers (Figure 1). At the bottom layer is the HDF5 file or other data source. Above that are two layers corresponding to the HDF library. First of these is a *low level interface* that concentrates on basic I/O: opening and closing files, reading and writing bytes, seeking, etc. Next is the *high-level, object -specific interface*. This is the API that most people who develop HDF5 applications use. This is where you create HDF objects such as a dataset or group, and perform operations on the objects, such as reading and writing datasets and subsets, etc. At the top are applications, or perhaps APIs used by applications. Examples of the latter are the HDF-EOS API that supports NASA's EOSDIS datatypes, the NeXus API that supports instrument definitions and corresponding data, and the DSL API that supports the ASCI data models.

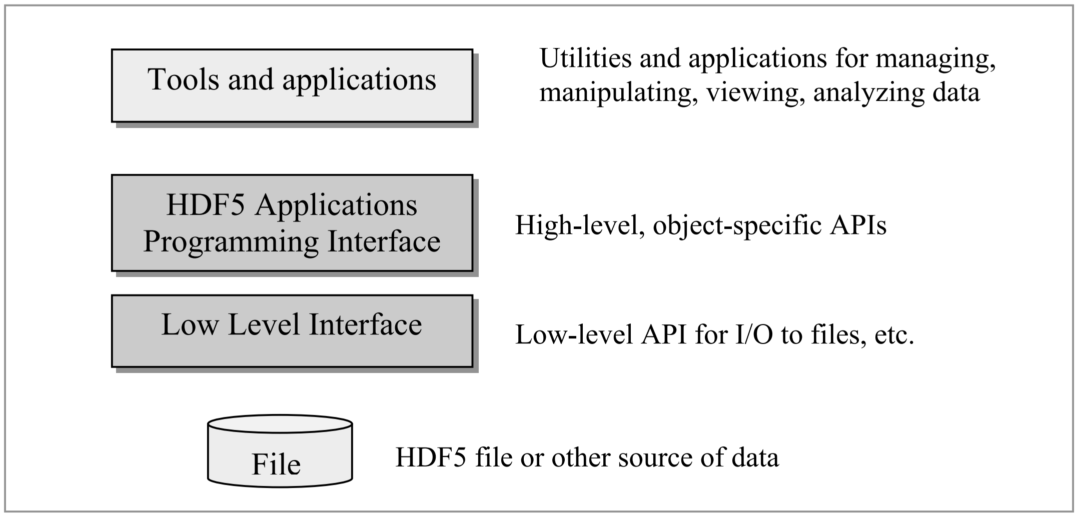
Tools and applications
Utilities and applications for managing, manipulating, viewing, analyzing data
HDF5 Applications Programming Interface
High-level, object-specific APIs
Low Level Interface
Low-level API for I/O to files, etc.
File
HDF5 file or other source of data

Figure 1. HDF library in context.

The way that HDF5 handles storage and I/O adds considerably to the complexity of the project. This is illustrated in Figure 2. When data is written, if frequently must be transformed, by changing its datatype, its endian-ness, by compressing it, and perhaps by storing it in chunks. The virtual file options make it possible to write and read in a variety of ways, including the normal standard I/O, MPI-IO (for parallel I/O), and a "split file" (metadata in one file, data in the other). This layer is exposed to users by a public API, so that users can write their own drivers for reading and writing to places other than those already provided with the library.

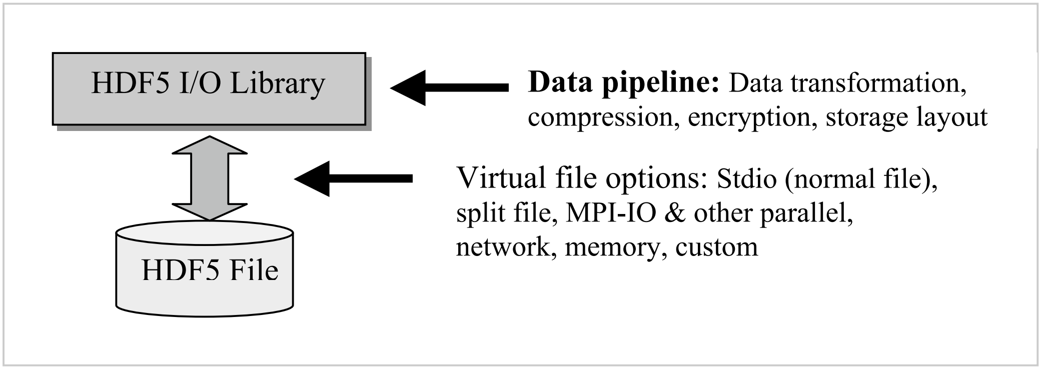
HDF5 I/O Library
Data pipeline: Data transformation, compression, encryption, storage layout
Virtual file options: Stdio (normal file), split file, MPI-IO & other parallel, network, memory, custom
HDF5 File

Figure 2. User controlled I/O and storage.

At the top level, the HDF5 library is accessible with a number of compilers and languages. The basic library is written in C, but included full wrappers for C++, Fortran90, and Java. The project supports vendors compilers on all platforms where they are available, and the GNU compilers wherever possible.

Because portability is an important requirement of HDF, a good deal of energy goes into supporting it on as many platforms as possible, sometimes on several different operating system versions on a single platform. A partial list of supported architectures and operating systems includes Sun Solaris 2.7 and 2.8 (32-bit and 64-bit), SGI IRIX6.5 and IRIX64-6.5, HP HPUX 11.00, IBM AIX 5.1 (32/64-bit modes), OSF1, FreeBSD, and Linux (SuSe, RH8, RH9) including 64-bit.

We see that HDF has many variations at many different levels, and this is what makes the project particularly challenging. Figure 3 illustrates the complexity of the project when taking these different options into account.

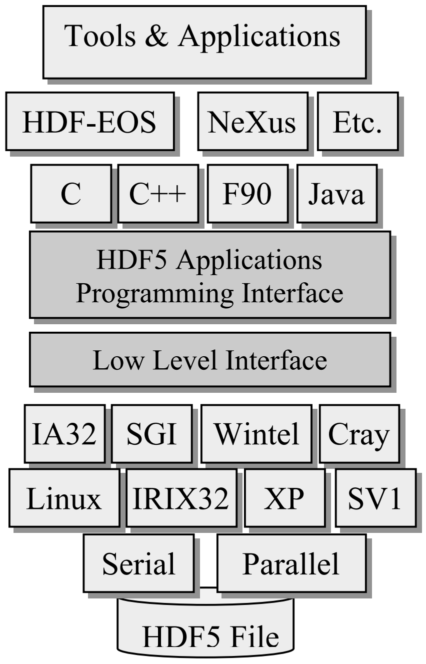

Figure 3. HDF5 library in context.

III. HDF statistics

In the previous section we gave a brief description of the HDF project. In this section we will try to give some “quantitative” characteristics of the project, i.e. what it takes to develop, support and maintain a project of such nature and size and why it is a challenging task.

As was mentioned above, the HDF group develops, supports and maintains several products:

- The original HDF file format and I/O library known as HDF4. Petabytes of NASA data from the Terra and Aqua satellites is archived and distributed in this format. The HDF4 distribution includes miscellaneous utilities to manage data stored in the HDF4 format.
- The recently-developed HDF5 file format and I/O library and utilities, which address the demands of high-performance computing and data storage.

- The H4toH5 Conversion Library and utilities to convert HDF4 files to HDF5 files.
- The H5Lite library, which provides HDF5 users with convenience API functions and APIs that enforce standard ways of storing images and tables in HDF5 files.
- Java interfaces to both HDF4 and HDF5 libraries and Java browser/editing tools to manage data in both HDF4 and HDF5 files.

The HDF5 projects alone consist of approximately 2073 files or about 917,000 lines of the source code. For all products this number exceeds 3,000,000 lines of code that is supported, maintained and distributed to HDF users all over the world. For the HDF5 Library, relative sizes of categories of the source distribution are shown below:

• C, C++ and Fortran source code	30%
• Documentation	30%
• Configuration code	15%
• Library tests	13%
• Source code for utilities	4%
• Tests for utilities	4%
• Examples and misc.	4%

The HDF group currently includes 15 full time staff members and 3 to 5 students (graduate and undergraduate). The group's annual budget is $2.1 million.

Most group members have been on the project for more than 7 years. Since the University of Illinois has one of the best Computer Science departments in the country, we do not lack bright, knowledgeable and hardworking students who are eager to help us with short-term development and maintenance tasks. Both factors play significant roles in the HDF group's ability to meet the demands of the constantly growing HDF users' community.

The sections below describe how the group operates, its mission, how it sets goals and achieves objectives, and the group's day-to-day practices that lead to the success of the HDF project.

IV. How do we measure success?

There are a number of components to how we measure success. Although we describe them here somewhat formally, in reality they have emerged over the years in an ad hoc and informal manner. The components include

- Mission
- Goals and objectives
- High quality, useful technologies
- Strong and continuing relationships with users

- Strong, committed development team
- Great working environment
- Adequate funding

IV.1 Mission, goals and objectives

The mission of the HDF Project is *to develop, promote, deploy, and support open and free technologies that facilitate scientific data exchange, access, analysis, archiving and discovery.*

Corresponding to the mission are a number of goals, including the following examples:

- Innovate and evolve HDF software and services in concert with a changing world of technologies
- Maintain a high level of quality and reliability
- Collaborate and build communities
- Build a team

For each goal, there are objectives, which are statements of how we intend to reach the goals. For example, the goal "Maintain a high level of quality and reliability" has the following objectives:

- Improve testing
- Implement a program to insure excellent software engineering practices
- Develop and execute a plan to meet quality/reliability standards

We think that the ability to constantly focus on the project's mission and to achieve its goals and objectives are vital to the success of the HDF project.

IV.2 Software quality and usefulness

The reason for HDF is to address users' needs and demands, both current and future. This means, for example, enabling users to deal effectively with big files, to access data in parallel, and to be able to handle large numbers of objects. We measure success in terms of the *HDF software's usefulness and quality*.

Usefulness of HDF can be measured by the number and types of applications that are able to use it, by the appropriateness of HDF APIs and data models, availability and effectiveness of tools, and interoperability with other software, including commercial tools such as IDL, MatLab, and Mathematica.

Acceptability is another measure of usefulness. The acceptability of HDF is attested to in those instances in which HDF has become a de facto standard. For instance, HDF has become an open standard for exchange of remote-sensed data, with over 3 trillion bytes stored in HDF and HDF-EOS, or megabytes of experimental data stored in the NeXus files.

We consider *HDF software quality* very broadly: it is not only the correctness and robustness of the HDF library. Two criteria described below are derived directly from the HDF mission and goals.

In the case of HDF, can data be shared across systems and across applications? Does the software run correctly and efficiently on all needed platforms? Thus *Portability* becomes a principle criterion for assessing HDF software quality.

Sustainability also is important, as many HDF users want to be able to commit to using HDF for a long time. Can a user read data written 15 years ago on an obsolete platform? Will the HDF software be available in 15 years?

IV.3 Relationships with users

In the end, users are the final assessors of the quality and usefulness of the technology and of the project itself. We measure the goal of "a strong and continuing relationship with users" by several measures:

- The number of users
- The number of *happy* users
- The number of *unhappy* users
- How well users achieve their goals by using HDF technologies
- How often users return with new needs
- The level of financial support we are able to receive from users

We see steady growth in the number of HDF users and in the number of the domains users come from. We work very hard to satisfy and address HDF users' needs. The following sections describe how we organize our project and what we do to "make HDF users happy."

V. How can we *achieve* success

We have seen that there are two general areas in which we measure success: by the quality and usability of the products, and by the satisfaction and involvement of users. Achieving success in these areas involves the following:

- An approach to needs identification, software design, and software implementation based on sound principles of software engineering
- Activities that maintain strong, responsible, and continuing relationships with users
- Effective technical processes for developing, testing, integrating, and maintaining software
- Business and social processes based on sound group management principles

The following sections provide detailed descriptions of these activities and processes.

VI. Developing and maintaining high quality, usable software

We have identified the following steps to developing software.

1. Discover and clarify need
    - (a) Discover need
    - (b) Identify sponsor
    - (c) Clarify need
    - (d) Enter into project plan, with initial estimates (time, sponsor, resources, priority, lead)
    - (e) Produce Request for Comment (RFC)
    - (f) Get feedback on RFC and revise as necessary
    - (g) Archive final version of RFC
2. Develop an implementation approach
    - (h) Assign task
    - (i) Create initial design or approach and generate "design/approach" RFC
    - (j) Get feedback on RFC and revise as needed.
    - (k) Develop validation plan – how will this be tested?
    - (l) Archive RFC
    - (m) Revise project plan according to RFC results.
3. Implement
    - (n) Implement, including tests.
    - (o) Ask sponsor to look at it and give feedback.
    - (p) Review result and repeat steps (d) – (h) as needed.
    - (q) Clean up: put in release notes, make sure it is covered in all appropriate documentation, and announce.
    - (r) Debug and revise as needed.
4. Maintain and support

VI.1 Discover and clarify need

Discover need. For a software product like HDF, there is always an abundance of suggestions for improvements and additional features. The problem/request may originate from a user, a sponsor, or one of our own team.

Identify sponsor. It is also important to identify the sponsor for any given need. The sponsor may be an organization that will pay to have the need satisfied, or it may be "everyone," when the need is some general feature that we know will help a large number of users.

Clarify need. Whoever it comes from, some effort is given to clarifying what a particular need is, as well as its importance. This can only happen successfully through close participation between the originator and the team. For instance, often a user identifies a need for a library change, only to find that a different approach to using the library makes this need unnecessary. For this reason it is important for team to have as much of an end-to-end understanding of the user's needs as possible.

With HDF, we try hard to attend meetings or workshops in which users describe their work. (Their *work*, not just their *needs*.) When possible, we follow up these meetings with presentations to the user groups, outlining the work that might be done. This helps clarify that the two parties have a good mutual understanding.

The interactions between requesters and HDF team cannot be a one-shot exercise. There has to be close collaboration with frequent interactions. The frequency of interaction depends on the product, but we try to have telecons with our most important users at least once per month. Sometimes the interactions are daily. Time must be allocated for this activity – in the long run it saves time by keeping a project on track and avoiding unnecessary effort.

Enter into project plan. Once a need is clarified and considered appropriate to do, it is entered as a task the HDF project plan. The HDF project plan is a large plan that contains all possible tasks that we might work on in the next year or so. We keep an additional plan of tasks that may be done someday, but are not seen on the horizon.

We try to do a project plan for anything that will take more than a week. We should do a project plan for shorter tasks, but we rarely get around to it.

One reason for a project plan is that it forces us to think through what we need to do. No matter how many times we develop software, we still tend to forget things that will need to be done, especially things that we would rather not do. Things like writing an RFC, developing a testing plan, writing documentation (ref manual *and* user's guide). Instead we generally think "how long will it take me to write my code?", and that's it.

Microsoft Project is used to organize the project plan. Although it takes a good deal of time to use and to keep current, it helps a great deal in understanding the complexities of the overall project, and in planning individual tasks. It is also a good device for communicating with team members about their responsibilities and commitments, and the role of their tasks in the overall project.

When a task is first entered into the project plan, it is usually not fully understood. Nevertheless, at this stage it is useful to prioritize the task, make initial estimates of the duration and resource requirements, identify a person who will be responsible for the task, and perhaps identify one or more persons who might work on it.

A project plan includes a work breakdown structure (WBS). The WBS part of the plan involves breaking each task into sub-tasks, estimating the duration of each, assigning sub-tasks to people and identifying dependencies. We use Microsoft Project for this. We also include these extra columns in our project plan:

- Sponsor – catch-all for who's paying for it, who asked for it, who is it important to)
- Lead – who is responsible for getting the task done. Often this is not the developer.
- Priority – what is the priority for this task. We have many priority levels, focusing largely on how important a piece of work is to a particular future release.
- Duration – how long the task should take. Making time estimates is a challenge. Like everyone else, we are poor at this, but some of the activities described above and below make us better than we might otherwise be. Developing software is always a learning experience, especially interesting software, so a lot of what developers do can only be known after they do it. That said, it is really important to keep trying.
- Release, etc. – what software release are we targeting, if any? This is a catch-all column, because some tasks are not for a particular release, but for some other purpose. Some tasks are ongoing – these need to be included so that we remind ourselves that the person doing them can't spend full-time on other things. We also have to remind the person of this.
- Notes – MS Project lets you attach a note to each task. We use this feature a lot. We find that it's easy to forget what was meant by a task, or some tidbit of information that clarifies what's important to consider, or a suggested approach, etc.

Needs/approach RFC. Once a task reaches a high enough priority that we plan definitely to work on it, someone is assigned the task of writing up a needs/approach RFC (Request For Comment). This describes the needs and general approach that is planned.

We try not to start a project until a needs/approach RFC has been written, but in practice we only do this under certain circumstances, such as (a) when we know there are a number of interpretations, (b) the task can be big or small, and we need to know what parts are most important, or (c) we don't have much knowledge in the area of the request and need to educate ourselves (and perhaps the sponsor). This RFC speaks to both the HDF team and requester, and requires both of their input.

Developers generally do not like to write RFCs. They want to get started with the real work – coding. Developers often have to be forced to write RFCs, and should be praised and rewarded for this unpleasant task.

Get feedback on RFC and revise as necessary. The needs/approach RFC is first circulated among team members, then revised and circulated among all who are deemed to have a possible interest in the work.

It is not enough to ask people to read and respond to RFCs. Some will read them, but most won't read them or won't read them critically. We find that we have to place most people in a situation where they *have to* respond. We have two weekly slots during which we discuss RFCs and other technical matters. People are expected to keep those times free. RFC review meetings help the team focus on the RFC and usually stimulate good questions and ideas. They always results in revisions to the RFC.

Ideally, we would engage the entire user community in this activity, but that is impractical, so we identify representatives and pigeonhole them to participate. It is important at this stage aggressively to seek comments from the community.

The comments from the meetings and other feedback are folded back into the RFC, and new versions are produced as needed. RFCs may have to go through several iterations. Developers dislike doing this, but they must. Praise them for it.

Archive final RFC. RFCs must be archived and not lost, and it should be standard practice to refer back to them when questions arise about the purposes of a particular task and the reasoning behind some of the decisions made.

VI.2 Develop an implementation approach

Assign task. At this point, there is a commitment to carry out the task, and one or more staff members are assigned to the task.

Create initial design or approach and generate "design/approach" RFC. If the task is substantial, it will probably require a design or approach to be worked out. This is done and written up in RFC-form. In some cases, this can be just an extension of the needs RFC, and in many cases they can be combined into one, but not always.

This RFC is generally circulated just among the team, but it's good if possible to have the sponsor (or others who would be influenced by the results), especially if the project is major or of particular importance. As with the needs RFC, it is important aggressively to seek comments from the team, and there is sometimes also a seminar to review the document. Comments are folded back into the RFC, and new versions are produced as needed.

Develop validation plan. In the case of substantial features, we also encourage a validation plan. That is a plan for how the feature will be tested. If the feature is a utility with many options, this plan can be very extensive. It is of course important to have a validation plan in order to test the feature after it is implemented, but a validation plan also frequently sheds light on ambiguities and other questionable aspects of the design.

Testing consumes considerable time and resources. Many believe that tests should be identified during the design phase (or earlier) before any coding begins. This includes both unit and system integration tests. We agree with that principle, but in our experience, we just haven't done this very well. For one thing, we don't have an official test enforcer, so they get forgotten if the project lead doesn't insist on it. That said, when we do follow this principle, we are always glad we did. Certainly, a lot of tests won't even be thought of until later because of changes in design and requirements that occur during implementation, but going through the exercise of identifying tests ahead of time is very valuable.

The testing plan may be included in the design RFC, or perhaps in a second document.

Archive RFC. Once the design is felt to be mature enough to begin implementation, the RFC and validation plan are archived.

Revise project plan according to RFC results. The project plan can now be updated to reflect a (presumably) more accurate understanding of the task and what it will take to carry it out.

VI.3 Implement

The following is a sketch of what we do during this phase. Many details about what is done during this phase are covered below under "Group Practices."

Implement. Implementation means coding, testing and documentation. Coding in the HDF group is generally done by an individual. Details about this phase are covered below under "group practices."

Ask sponsor to review and give feedback. Every effort should be made to get feedback from sponsors during this phase. Because we check in code changes to a "development branch" of the library, we are able to make this version available to sponsors who have a particularly strong interest. This helps achieve timely feedback and can avoid misunderstandings. Ideally sponsors provide us with code to test whether our library changes meet their needs.

Release. As with most software development, the release process can be very extensive. Documentation must be completed, all platforms must be tested, and the release announced. A release process can last from a couple of weeks to three months, or more. Most often it takes about two months.

VI.4 Maintain and Support

This is the most important part of the software development process for our users, though it is often the least exciting and most time consuming part of the process for the development group. During this stage, we port to new compilers and operating systems, tune for performance, fix bugs, and add documentation. We may also repeat steps 1-3, as described above, based on user feedback.

VII. Group practices – technical

Source code statistics. The HDF software source code statistics were described in section III. As you can see, there is so much code that individual team members must be assigned to support particular parts of the HDF libraries, utilities, or documentation**.**

Release levels. Three types of release are available:

1. *Development release and snapshots.* As noted, the release under development is made available frequently as a "snapshot", allowing daily testing (see "testing" below) and making the latest code available to friendly users and sponsors.
2. *The official release*. This is the release that has undergone thorough testing, is presumably very stable, and that users are supposed to rely on.
3. *Past releases.* It is very important to recognize the release of new versions of software can be very disruptive on users of the software. Some HDF users have very elaborate software systems themselves, and cannot change releases on the same schedule that we do. For some, adapting to a new release may be impossible. For

these reasons, past releases are extremely important, must be readily available, an need to be supported fully.

Some official releases are "minor" in that they involve only small changes to the library, and often some performance improvements. Minor releases occur about 2-3 times per year. A "major" release is likely to contain substantial changes to the API, and may even contain changes in the underlying format. Historically, these have occurred about once every two years. It is usually important to synchronize releases with users.

We follow the numbering system used by Linux and other developers: odd-numbered releases are under development, and even-numbered releases are final supported releases.

Source code management. Everyone uses CVS, and the use of CVS is well-defined, with conventions reasonably well documented and enforces. (E.g. check-in includes information about how code was tested.) Serious effort is given to determining how best to package code – and this evolves over time.

Coding standards. This is a tricky and sometimes nuanced issue. In general, coding standards are encouraged and enforced. At the same time there has to be some freedom for differences – platform, compiler, and IDE differences have to be accommodated, and individual stylistic differences can be important.

Code reviews. We would like to do code reviews, but have found that we just don't have the time for them. In some cases a senior person will review the code of someone who is new to the library or tool, and that can be beneficial. Such review, however, must be done with some tact, because coding can often be a very personal activity.

Maintaining platform-independence. The group has developed good knowledge of what it takes to achieve platform independence. There are many aspects to this, from coding guidelines to the overall design of the product. A lot of resources are required to maintain the configurations and build environments that support testing and platform independence.

Maintaining time-independence. It is important to make sure the code holds up over time, because others rely on it. Every user of HDF5 has a different time frame for upgrading to new releases of the library. For instance it is usually necessary to support versions of the library for several different versions of certain architectures, operating systems, compilers and file systems. Supporting these different versions adds enormously to the cost of supporting HDF.

It is also important that the source code itself hold up as long into the future as possible, because many users archive their data in HDF and will need to access it into the future.

Rules for changing APIs. This is still a difficult, tricky issue. Different users have different needs, and it is hard to balance the need for a stable API and library (e.g. for vendors and other users who can't easily rewrite code to accommodate changes) against the need to respond to new requirements and shortcomings that are discovered in the code.

Testing. Testing is absolutely critical to the project, and occupies probably 15% of all activities of the group. Tests currently make up 17% of the sources code, and should make up a larger percentage. Testing occurs in a number of ways in the project:

1. *Testing before code check-in*. Any new code, including code changes (e.g. bug fixes), should be tested on three different platforms before they are checked into CVS. Some developers are really good at this, and others can be more forgetful. We've tried various mechanisms to enforce this, but nothing very draconian.
2. *Daily regression testing*. We have automated daily testing of the entire code base on a number of platforms. There is a "platform watcher" assigned to each platform, who is supposed to notice when any tests fail on their platform, and try to suggest some action. In practice, there are a couple of people who do most of the watching. Since daily testing takes a lot of resources, it only occurs if changes have been made in the code.
3. *Weekly regression testing*. A second tier of platforms are tested weekly, also automatically. These are perhaps less important platforms, or platforms that are harder to reach or for some other reason not good for daily tests.
4. *Remote system testing*. There are some key remote systems that we test regularly. These are provided by vendors and others whose users need HDF
5. *Testing of different configurations.* Figure 4 illustrates the enormous number of combinations of library features, languages, architectures, operating systems, and file systems that HDF5 needs to support. Extensive efforts are made to cover as many different configurations as possible.

THE TESTING CHALLENGE

machines x operating systems

x compilers x languages

x Szip (encoder + no encoder)

x (serial + parallel)

= a very large number

Figure 4. The testing challenge. HDF5 is currently tested daily in 68 configurations, HDF4 in 14.

Documentation. We have a full-time documentation person, plus a part-time student. Also, the QA lead and other staff participate significantly in documentation. Compared to most groups, we are rich in this area. Still, this is far too little to meet our needs, and we are reminded of this almost daily.

Rapid prototyping. This is tremendously valuable when we do it, but we don't do it very often.

VIII. Group practices – business and social

Staff breakdown. As we described in section III**,** the HDF Group at NCSA has 15 full-time staff, plus between 3 and 5 students, supported by an annual budget of approximately $2 million. We have staff for user support, documentation, quality assurance, software development, testing, and team leadership. There is a lead for tools, QA, library development, weird platforms, high performance systems and applications, and system administration). Team leads have different skills and emphases – they complement one another and keep each other honest. We have inadequate staff in some of these areas.

In an effort to determine how technical staff members spend their time, we asked each member to estimate how much time they spent on a variety of activities. These results are gross estimates, but they do give a sense of how we distribute our time.

Code development	24%
Test writing	4%
Pre check-in testing	3%
Release testing	5%
Platform support	4%
Peer-to-peer communications	14%
User's support	17%
Docs, RFC, consulting	17%
Other comm. with users	2%
Meetings, etc.	11%

Summarizing, according to these estimates, software implementation, testing, and internal communication consume about 54% of technical staff time, and support-related activities about 36%. Approximately 10% of staff time is devoted to other purposes.

Accountability of everyone to the whole process. We strive for an attitude that we are all, as a team, accountable for the end product. There should be no handing off in the sense of "it's your problem now, not mine." One way this is emphasized is that every developer participates in testing – everyone has responsibility for one or more platforms. This spreads the knowledge and accountability.

Help desk. We have a 3/4-time helpdesk who is very well-organized and good at working with users. Also very important is the existence of a strong relationship between the helpdesk and developers. The helpdesk serves as a clearinghouse – often it can answer questions, but sometimes it needs to prioritize requests, and recognize when to consider fast-track responses to problems. This can be a distraction for developers and others, and often leads to delays in other parts of the schedule. It is a balancing act. It is very important to be aware of the phenomenon and constantly monitor it and react accordingly.

Approaches to carrying out tasks. Every task is different, so there are variations on how every task is dealt with. Difficulty, urgency, priority, cost, and availability of expertise are the factors that usually define our approach to the task. All combinations of these factors can occur, so the way they are treated varies enormously. Unless you have a team that is pulling together and understands one another, it is impossible to deal with all of these effectively. Having agreed-upon processes helps a great deal, even if you don't always follow the same processes.

Weekly HDF5 developers' meetings. In meetings, structure is important, as is consistency – they help insure that things get covered and get done. Consistency is also important because it sends a message that we believe in the process and are serious about it.

Meeting notes include notes of all of the items shown below. They are displayed on a screen during the meeting, and made available on a web site. Meeting notes are important in both remembering and clarifying decisions.

The meeting structure includes:

(a) Set agenda. In particular, recommend topics for technical discussion. See (d) below. We encourage people to identify these discussions before the meeting, and have a rule that nobody should be expected to read background material without receiving it two days before the meeting.

(b) Review action items. A running list of open action items is kept. This is displayed and each item is reviewed.

(c) Reports. Each participant submits a report prior to the meeting, describing work done since the last meeting, with the goal that the whole team is aware of what each person is doing, what issues they are addressing, and what results they are getting. Technical discussions often arise during this time, and these are not allowed to go on for very long. If further discussion is called for, this is identified and planned.

(d) Technical discussions.

HDF seminars. A two-hour time slot set aside for weekly seminars, presentations of technical papers, lectures on the innards of HDF, and discussions of important issues. This is valuable in facilitating communication and distributing knowledge in the group.

Decision-making processes. Ideally, decisions are never made without consultation, but this is a "herding cats" problem. The HDF5 developers meeting is a critical part of this. After a certain amount of debate, if an issue is not resolved, it goes to a single person or a committee. Generally, we believe that it is better to take the necessary time to make good decisions than it is to make decisions quickly.

Frequent meetings with staff and performance reviews. Supervisors meet frequently (usually at least weekly) with staff to assess and plan activities. Performance reviews serve a similar purpose (among others), in that they can determine how individuals are meeting their areas of responsibility, being very specific about what those areas are, and setting metrics for determining success.

IX. More about project planning

Who are the sponsors? As mentioned above, every task is assigned a sponsor. Sponsors include organizations that provide funding, users whose applications help promote the use of HDF or make it widely available, and someone in the group who champions implementing a particular feature or other task.

Re-prioritization and re-definition exercises. It is important frequently to revisit the project plan. The HDF "to do" list has over 500 items, with perhaps 100 that are considered high priority, and maybe 10 that are being worked on at any time. (This doesn't include "ongoing" tasks, which are also factored into the project plan. This is important when estimating how much time a task will take.)

Project classification. Projects are classified as one of the following. The project plans divide them into these categories so that we and our sponsors are in agreement with respect to expectations.

(a) *Outreach* – HDF workshops, HDF users conferences, face-to-face meetings with current and potential HDF user

(b) *Documentation* – Reference Manual, User's Guide, tutorials

(c) Testing

(d) Performance studying and tuning

(e) *Research* – try out a feature, write a report, share expertise gained with others. Outreach is very important in this case. This work may be used later on, but at this stage we're just trying to learn and help our sponsor learn.

(f) *R&D* – Work that we intend to use, if it works out.

(g) *Development* – Work that we know will be used and must be made to work. Typically there is a specific user's need and resources are paid for. It has seemed convenient to put "development" work into two categories:

- Technology infusion, where the results of a research or R&D project seem interesting enough to implement.
- Library and tools enhancement. This includes bug fixes and addition of new features or performance capability.
  Tools example: someone needs the h5dump tool to display a certain kind of data in a different way.
  Library example: someone needs to be able to convert integers to floats when reading from an HDF5 dataset.

Technology readiness. The development of software often occurs in a number of stages, each subsequent stage corresponding to an increasing level of maturity, completeness, and robustness. NASA's Technology Readiness Levels (see Appendix 1) provides a good frame of reference for thinking about this. Precise levels are not important, but the concept of readiness is. The TRL document helps remind us of what this means.

X. Summary of strengths and challenges

The strengths of the HDF project are described in some detail in the previous sections of this document. A summary of the strengths of the HDF project would include

(a) Emphasis on user support, outreach and high quality documentation.

(b) High quality and diverse staff with good morale and commitment.

(c) Ability to address all aspects of product development, emphasizing quality control, fast bug fixing and frequent releases, and the ability to focus on a single product (or two) over a long term.

(d) The project sponsors, who provide a high level of support, excellent user feedback, and broad visibility.

Challenges faces by the project have not been described as fully, and will only be summarized here. Challenges fall into several categories

(a) The software development team can improve in certain ways.
   i. There is insufficient distribution of expertise: certain areas of expertise are currently concentrated among too few staff.
   ii. There are communication challenges: the group has to work continually to achieve adequate communication.

(b) Software development processes can be improved in a number of ways.
   i. Configuration and porting take a great deal of time
   ii. Although testing is extensive, the amount of testing currently done is a small fraction of what could and should be done.
   iii. The release processes take great deal of time
   iv. Maintenance takes more resources than the project would like
   v. We would like to do more prototyping
   vi. The need to work well on existing technologies, can make it difficult to keep up with new technologies
   vii. Supporting parallel I/O in a growing number of environments is hard.

(c) Usability: many users and potential users have difficulty using HDF.
   i. There is a tradeoff between flexibility and ease of use for casual users. HDF5 in particular needs to simplify its use.
   ii. Although the documentation is relatively good for free software, but is still insufficient for many users, given the complexity of HDF5 and the challenge of using it optimally in any particular situation.
   iii. There are not enough tools for high level users, especially casual users who have no previous knowledge of HDF.
   iv. Common tools need to be better able to read, and sometimes write, HDF4 and HDF5. Also, ways need to be found to convert between the two HDFs and other common formats that deal with similar data.

(d) Marketing: the HDF technologies have potential applicability well beyond its current uses.
   i. More energy should be expended in making HDF known to potential users.

ii. The project needs to find ways to connect with users and potential users.

iii. There needs to be an "HDF book" describing fully the philosophy behind HDF, the HDF technologies, and the applications that use HDF.

(e) Viable long-term support: The project has very good support from certain agencies, but long-term support is not guaranteed.

i. A business model needs to be developed that will lead to the sustainability of HDF.

ii. We need to provide an excellent working environment to retain our current staff members and attract new ones in order to address the needs of the HDF user community.

(f) Usability: the software is too hard to use, insufficiently documentation and tools.

(g) Marketing: the software is inadequately marketed

(h) We don't to enough prototyping.

(i) System administration: too much of this has to be done by staff, and half-time sysadmins have not worked out well so far.

1. Biggest areas where we would like to put more resources.

(a) Configuration and build.

(b) Marketing.

(c) Reporting, particularly technical reports and periodic reports to sponsors.

Acknowledgments

The authors would like to thank all HDF group members for the fruitful discussions that lead to this paper, especially Bob McGrath for helping with the HDF source statistics and Frank Baker for securing our victory in "HDF People v MS Word and PDF."

Appendix – NASA TRL

NASA DEFINITION OF TECHNOLOGY READINESS LEVELS

TRL 1 Basic principles observed and reported

Transition from scientific research to applied research. Essential characteristics and behaviors of systems and architectures. Descriptive tools are mathematical formulations or algorithms.

TRL 2 Technology concept and/or application formulated

Applied research. Theory and scientific principles are focused on specific application area to define the concept. Characteristics of the application are described. Analytical tools are developed for simulation or analysis of the application.

TRL 3 Analytical and experimental critical function and/or characteristic proof-of-concept

Proof of concept validation. Active R&D is initiated with analytical and laboratory studies. Demonstration of technical feasibility using breadboard or brassboard implementations that are exercised with representative data.

TRL 4 Component/subsystem validation in laboratory environment

Standalone prototype implementation and test. Integration of technology elements. Experiments with full-scale problems or data sets.

TRL 5 System/subsystem/component validation in relevant environment

Thorough testing of prototype in representative environment. Basic technology elements integrated with reasonably realistic supporting elements. Prototype implementations conform to target environment and interfaces.

TRL 6 System/subsystem model or prototype demonstration in a relevant end-to-end environment (ground or space)

Prototype implementations on full-scale realistic problems. Partially integrated with existing systems. Limited documentation available. Engineering feasibility fully demonstrated in actual system application.

TRL 7 System prototype demonstration in an operational environment (ground or space)

System prototype demonstration in operational environment. System is at or near scale of the operational system, with most functions available for demonstration and test. Well integrated with collateral and ancillary systems. Limited documentation available.

TRL 8 Actual system completed and "mission qualified" through test and demonstration in an operational environment (ground or space)

End of system development. Fully integrated with operational hardware and software systems. Most user documentation, training documentation, and maintenance documentation completed. All functionality tested in simulated and operational scenarios. V&V completed.

TRL 9 Actual system "mission proven" through successful mission operations (ground or space)

Fully integrated with operational hardware/software systems. Actual system has been thoroughly demonstrated and tested in its operational environment. All documentation completed. Successful operational experience. Sustaining engineering support in place.