ESDS-RFC-007v1
Category: Recommended Standard
Updates/Obsoletes: None

Mike Folk, Elena Pourmal
January 2007
HDF5 1.6 Standard

HDF5 Data Model, File Format and Library – HDF5 1.6

1 Status of this Memo

This is a description of a Draft ESE Community Standard.

Distribution of this Draft ESE Community Standard is unlimited.

2 Change Explanation

This is the initial published version of this document.

3 Copyright Notice

4 Abstract

This document defines HDF5, a data model, file format and I/O library designed for storing, exchanging, managing and archiving complex data including scientific, engineering, and remote sensing data.

TABLE OF CONTENTS

5 Introduction

HDF5 was created to address the data management needs of scientists and engineers working in high performance, data intensive computing environments.

Scientific and engineering applications must deal with increasingly large datasets, a variety of different datatypes and structures, and many different forms of metadata. This data must be available on a broad range of operating systems and computing environments. It must be easy to move data from place to place, and to share the data among widely differing applications.

Applications and tools must be available for creating and accessing the data, and the basic software for storing and accessing the data must work in as many different computing environments as possible.

Because the amount of scientific data is increasing rapidly, efficient storage is critical. Many applications require extremely efficient and flexible I/O in order to deal with the volume of the data being processed, the frequent need to access small subsets from large data sets, and the variety of architectures and file systems in use.

5.1 What is HDF5?

HDF5 consists of three major components: (1) a general-purpose data model, (2) a file format, and (3) an I/O library.

The HDF5 *data model* provides structures and operations to allow creation, storage, and access to almost any kind of scientific data structure or collection of structures. In addition to the HDF5 file object, the data model includes two primary objects (datasets and groups), a number of supporting objects (e.g., attributes and datatypes), and metadata describing how HDF5 files and objects are to be organized and accessed.

The HDF5 *file format* describes how HDF5 data structures are represented in storage, in memory, or on other media. Because HDF5 is designed for managing large data objects and complex heterogeneous collections easily and efficiently, the HDF5 format allows for alternate representations of many objects. The format is self-describing in the sense that the structures of HDF5 objects are described within the file.

The HDF5 *I/O library* implements the data model in a number of programming languages, including C, Fortran, C++, and Java. These API's are designed for flexibility – they give applications full access to available HDF5 storage structures and provide features for tuning applications for particular platforms, storage requirements, or I/O access patterns.

Complementing these three technical components are the approaches of the HDF5 project to intellectual property and community standards.

The HDF5 library, which is owned by the University of Illinois, is open source, and the HDF5 copyright[1] allows it to be used at no cost by all applications, including commercial applications. The HDF5 project and its sponsors work closely with vendors and non-commercial applications developers, to enable their products to support HDF5 effectively and to make sure that HDF5 meets the demands of these products for quality and performance.

As for community standards, the HDF5 project and its sponsors dedicate significant resources to developing, supporting, and enforcing standard uses of HDF5. Adherence to standards makes it possible to share data easily, and to build and share tools for accessing and analyzing data stored in HDF5. Standardization activities include establishing conventions for the use of HDF5 for particular applications. For example, HDF-EOS defines a data model built for earth science data, and the HDF-EOS API implements that data model. As another example, the HDF5 project defines standard ways to store raster images, tables, and other complex objects in HDF5, and provides high-level APIs to encourage adherence to these standards.

5.2 Motivation for Proposing Standardization

HDF5 is the underlying format for HDF-EOS 5. HDF-EOS is the standard format and I/O library for the Earth Observing System (EOS) Data and Information System (EOSDIS). EOSDIS is the data system supporting a coordinated series of polar-orbiting and low inclination satellites for long-term global observations of the land surface, biosphere, solid Earth, atmosphere, and oceans. HDF-EOS 5 is the standard for the Aura mission, which is the final EOS mission.

HDF5 is also to be the distribution format for the National Polar Orbiting Environmental Satellite System (NPOESS), a satellite system to be used to monitor global environmental conditions and to collect and disseminate data related to weather, atmosphere, ocean, land and near-space environments.

EOS data stored in HDF-EOS 5 and NPOESS data to be stored in HDF5 are of fundamental importance to current and future research on global climate change and to scores of other applications of national and international importance.

ESE standardization of HDF5 will help to accelerate its adoption among the EOS and NPOESS communities, and many others as well, both through an increase in the number of developers writing to the specification and using the I/O library, and through an increase in the number of projects providing data in HDF5.

ESE standardization will validate HDF5 to vendors of software that is important to users of EOS and NPOESS data, increasing the likelihood that these vendors will support HDF5 in their products.

ESE standardization will validate HDF5 to government agencies and other organizations that have considered standardizing on HDF5 but have been reluctant to do so because of the lack of

[1] See Appendix E for the HDF5 copyright.

standardization. This will increase the likelihood of adoption of HDF5 by these organizations, in turn broadening the support for HDF5. Greater support and adoption of HDF5 would very likely result in improved usability of HDF5, more software for working with data in HDF5, and more useful data stored in HDF5.

Finally, ESE standardization will pave the way for broader standardization of HDF5, both at the national (ANSI) and international (ISO) levels. If this happens, the results of standardization described above will extend even more broadly, across a broad range of science and engineering domains. Many of these results will return dividends for the ESE community as well.

6 HDF5 Data Model

6.1 Introduction

The HDF5 data model describes the organization and structure of data stored in a computer system and specifies operations that can be performed on that data.

An HDF5 application program maps its data structures to the HDF5 objects defined by the data model. This mapping is independent of the storage medium, computational system and computational environment.

The potential complexity of an application program's data objects, and the richness of data structures and datatypes required to describe the objects, may present a challenge for any data model. The HDF5 data model uses arrays of structures to describe object data and metadata and a grouping mechanism to describe relationships between objects. HDF5 has proved suitable for describing data objects in scientific and engineering applications.

This chapter defines the HDF5 objects, metadata that is required to fully describe the HDF5 objects (also called structural metadata), and operations on the HDF5 objects.

UML diagrams in this document are based on the standards described in *UML Distilled: Applying the Standard Object Modeling Language* by Martin Fowler and Kendall Scott (ISBN 0-201-32563-2, Addison-Wesley, 1997).

6.2 HDF5 Objects

The HDF5 data model defines seven objects: file, group, dataset, link, datatype, dataspace, and attribute. All objects are subclasses of the class HDF object, as illustrated in Figure 1:

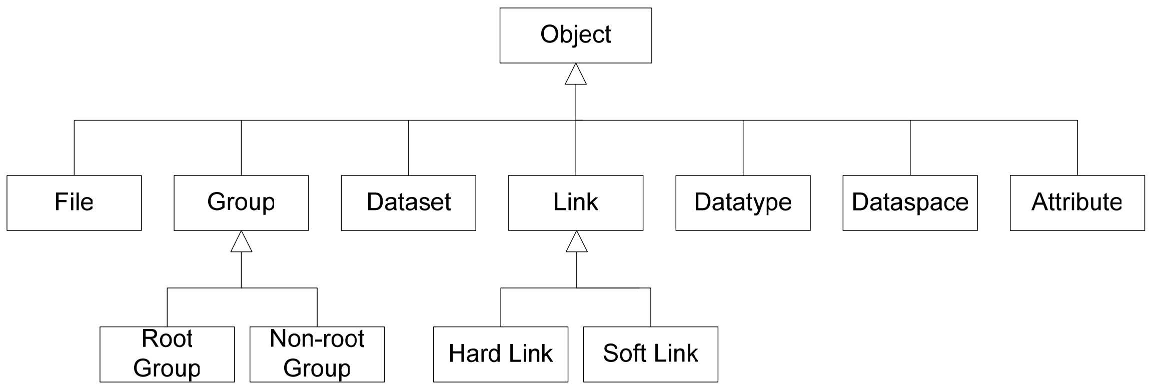


Figure 1. There are seven classes of HDF5 objects.

A *file* is a container for HDF5 objects.

Group and *dataset* (or data) objects are called the primary objects in an HDF5 file.[2]

A dataset contains an array of data elements, together with supporting metadata. *Dataspace* and *datatype* objects are necessary parts of a data object definition. Dataspaces describe the rank and dimensions of a data object array. Datatypes describe the data elements in a data object array.

HDF5 datatypes can be very complex. Multiple datasets stored in a file may have the same datatype. In order to avoid redundancy in describing the datatype for each associated dataset and reduce the size of dataset structural metadata, the complete description of the datatype can be stored in an HDF5 file and referred to when needed. Thus a datatype may also be a primary object (called a named datatype) along with being a part of data object structural metadata.

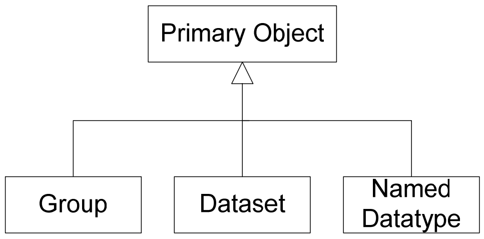


Figure 2. Primary HDF5 objects are individually addressable.

Any primary object has at least one hard link associated with it as shown in Figure 3.

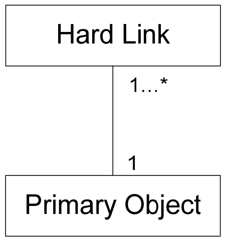


Figure 3. Hard and soft links can be associated with a primary object.

[2] In the HDF5 file format, a primary object is an object that has an absolute address within a file associated with it. A complete description of the object, including its structural and application-defined metadata and its raw data, can be found by decoding information stored at that file address.

Groups and links impose an organization among the objects in a file (see section 6.3.1for further discussion).

An *attribute* is a means of attaching content metadata to an object. Content metadata (also called user-defined or application-defined metadata) describes the nature and/or the intended usage of the primary object. It is not a part of the HDF5 object definition and is interpreted by application only. Figure 4 shows the association between primary objects and attributes.

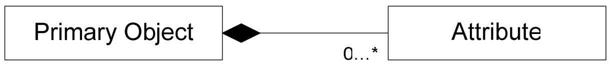


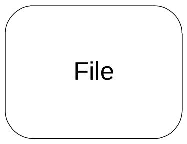
Figure 4. Any primary object, including the HDF5 root group, may have attributes.

Links provide a mechanism for locating the members of groups.

Sections 6.3 through 6.9 define each of these HDF5 objects, describe the object's structural metadata, and list the operations that can be performed on each object. Additional information is included where appropriate.

6.3 File Object

6.3.1 The HDF5 file



The HDF5 data model defines a file as a container for HDF5 objects. The concept of a file as a container for HDF5 objects is illustrated in Figure 5, below. In this example, the HDF5 file contains two group objects, two dataset objects, and one datatype object, all organized in a hierarchical structure.

Groups and links are used to organize objects in a file as a directed graph with a single designated entry node, called the root group, as defined in Figure 5 and illustrated in Figure 6 and Figure 7.

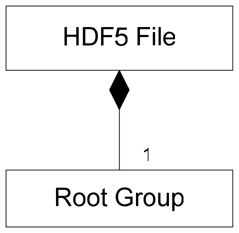


Figure 5. An HDF5 file contains exactly one root group.

Each edge in the graph is represented by a link object. Each node is represented by a primary HDF5 object. Internal nodes (nodes with outgoing edges) are groups. Terminal nodes (nodes with no outgoing edges) can be any primary HDF5 object (i.e. group, dataset, and datatype objects).

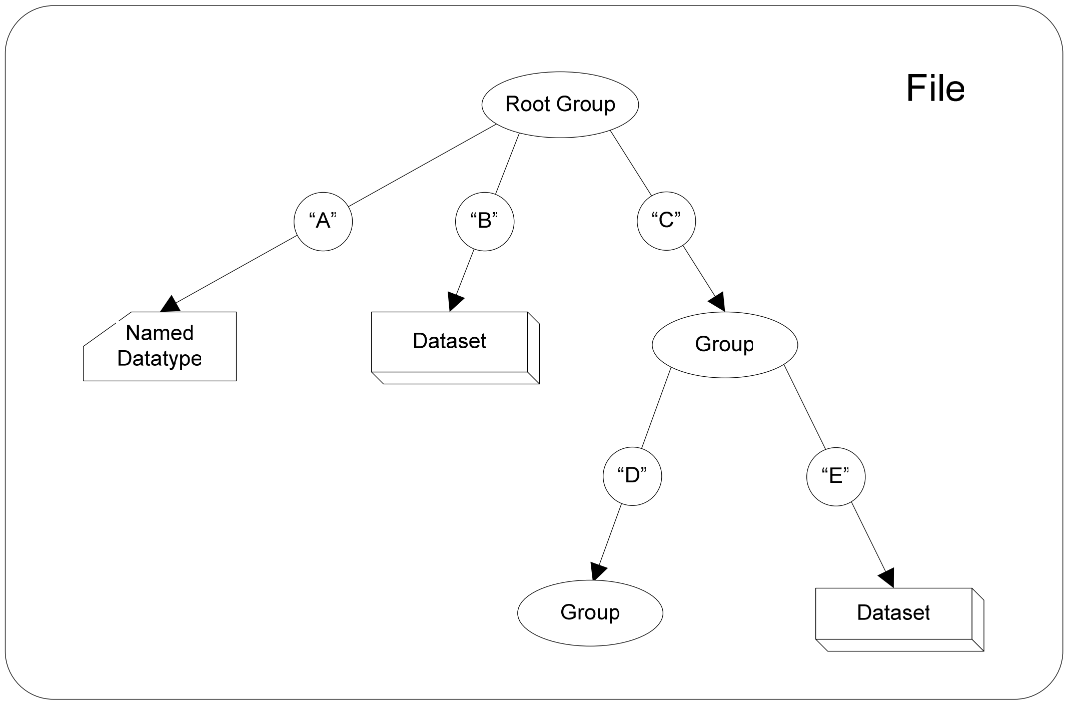


Figure 6. The HDF5 file serves as a container for HDF5 objects organized in a hierarchical structure. The arrows represent links and are the edges of the graph; groups, datasets, and named datatypes are the nodes**.**

An object in a file can be identified by any of the paths, consisting of one or more links, that connect the object with its ancestor nodes. The root group is identified by the path "/", a single slash. A path that begins with the root group is an absolute path; a path that begins with any other parent group is a local path. In Figure 6, the dataset in the lower right can be identified by the local path "E" under the group "/C", or by the absolute path "/C/E".

As illustrated in Figure 7, an HDF5 file may contain cycles. The root group, in this case, can be identified by two absolute paths, "/" and "/C/D/F", and by several relative paths. Links "C", "D", and "F" create a cycle in this graph.

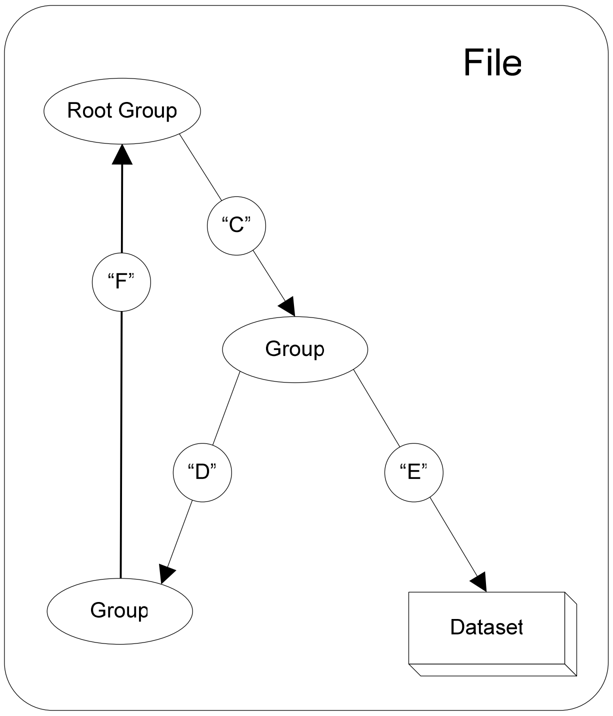


Figure 7. An HDF5 file hierarchy, being a directed graph rather than a strict tree, may contain cycles.

6.3.2 File structural metadata

File structural metadata is described in Appendix A, *HDF5 File Format Specification*, in section II, "File Metadata."

File structural metadata sets certain file object properties and cannot be modified after the file has been created.

File structural metadata contains version information and parameters of internal data structures used for storing HDF5 objects. It also contains information about how a file is accessed for different file storage layouts on a computer system. For more information on file access alternatives, see section 8.3.3, “Virtual file layer (VFL),” in this document.

HDF5 1.6 specifies the following elements of file structural metadata that are defined by an application program:

Name	***Description***
Size of offsets	Number of bytes used to store addresses in an HDF5 file
Size of lengths	Number of bytes used to store the size of an HDF5 object
Group leaf node K	Each leaf node of a group B-tree will have at least K entries but not more than 2K entries; a single leaf node may have fewer entries
Group internal node K	Each internal node of a group B-tree will have at least K entries but not more than 2K entries; a single internal node may have fewer entries
Index storage internal node K	Each internal node of an indexed storage B-tree will have at least K entries but not more than 2K entries; a single internal node may have fewer entries
Driver information	Information about a file driver needed to reopen an HDF5 file for different storage layouts (see section 6.3.2.1)

6.3.2.1 HDF5 file storage layouts

The HDF5 file format specification (see Appendix A, *HDF5 File Format Specification)* defines valid types of physical storage layout. For instance, a file can be stored as a single file or as a set of files, or it may exist only in an application program’s memory. The format also allows applications to define alternate forms of physical storage beyond those described in the specification.

HDF5 1.6 provides the following predefined storage layouts for a file object:

Storage layout	***Description***
Single file	A single file on a file system
Family of files	The logical space of an HDF5 file is partitioned among a set of single files that have the same size; this layout makes it possible to store HDF5 files larger than 2GB on a file system that does not support large files

Multi file	A set of single files that represent an HDF5 file; each file holds one or more predefined types of internal structures (global and local heaps, B-trees, objects headers, raw data) used to describe objects in an HDF5 file
Core (memory) file	An HDF5 file stored in the memory of an application program

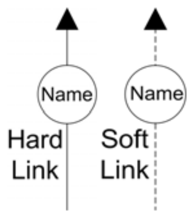
6.3.3 Operations on a file

The following operations can be performed on a file:

Name	***Description***
Create	Creates a file with defined structural metadata elements.
Open	Opens an existing file.
Reopen	Reopens an already open file.
Close	Closes a file.
Mount/unmount	Includes/removes one file’s graph structure in/from another file’s graph structure.
Get properties	Gets structural metadata of a file.
Query “Is HDF?”	Determines whether a file is an HDF5 file.

6.4 Link Object

6.4.1 The HDF5 link



A link represents an edge in a file graph structure – it connects a group with some HDF5 object, and as such is the means by which group membership is described.

The object that a link connects a group to is called the link’s *target*. A target may be any primary object, i.e. a group, dataset, or named datatype object.

The information associated with a link is stored in a group’s structural metadata (see section 6.4.2).

A link has a (name, value) pair associated with it. The *name* is a label that identifies the link. The name is an ASCII null-terminated string, and is used as a key for identifying objects in a file. A link’s *value* is used to resolve the link – to locate the link’s target.

There are two types of link objects in the HDF5 data model: soft links and hard links, which are defined and illustrated in Figure 8 through Figure 10.

Hard links are the primary means by which objects are associated with groups in an HDF5 file and are required in order to traverse the graph structure of an HDF5 file.

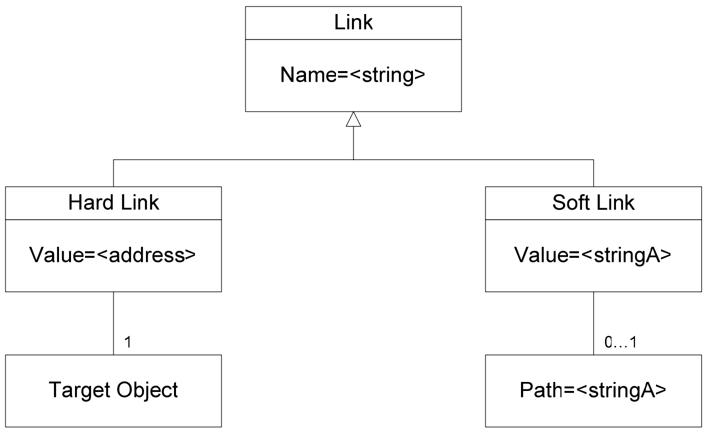


Figure 8. A hard link is always associated with exactly one primary object; a soft link may only be associated with a path in the file.

The value portion of a hard link's (name, value) pair is the file address of the object pointed to by the link. Thus a hard link can be created only when there is an existing primary object for it to point to.

The process of unlinking disassociates a hard link from its target object in the file. An object is accessible in an HDF5 file as long as at least one hard link that points to it still exists. An object that is pointed to by hard links maintains a count of the number of hard links that point to it. When a hard link is removed, that counter is decremented, and when the last hard link to an object is removed, the object is effectively unreachable once it is closed, and is considered to be deleted. Conversely, a hard link cannot "dangle" – when an object is deleted, all of the hard links that point to it must be deleted.

Unlike a hard link, a *soft link* does not point directly to an object. Rather it contains as its value a string, which is interpreted as a path that is used to find a hard link (or another soft link).

When a soft link is removed, the object to which it pointed is unaffected. Thus, soft links can be removed with no danger of accidentally removing the target object. Soft links are also useful for describing structure in an HDF5 file before the objects that constitute the structure actually exist.

If the target of a soft link is removed by removing one of the hard links in its path, the soft link is unaffected. Hence, it is possible for soft links to "dangle."

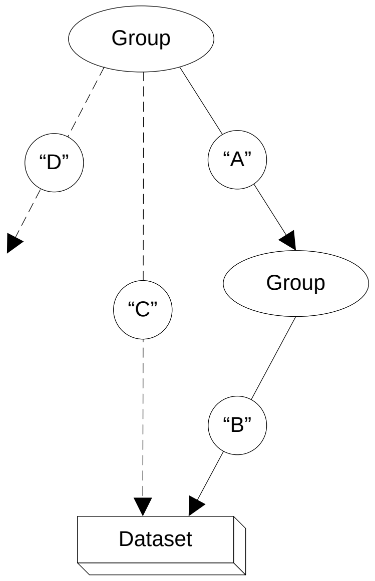
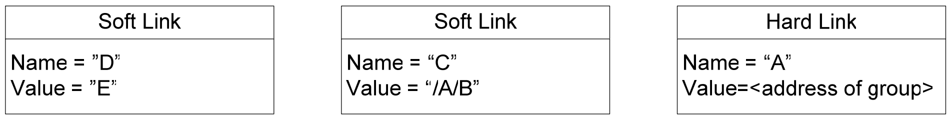


Figure 9. A hard link points to a primary object while a soft link may point to no existing object at all.

In the hierarchy illustrated in Figure 9, the top group contains three links: a hard link, "A", and two soft links, "C" and "D". The value associated with the hard link "A" is the address in the file of the group the link points to. The value associated with the soft link "C" is the path "/A/B"; i.e., this link points to the existing dataset identified by the path "/A/B". The value associated with the soft link "D" is the path "E"; this soft link is said to "dangle" because its value is a string specifying a path that cannot be associated with an existing object. Figure 10 shows the structural metadata (see section 6.4.2) of the link objects in Figure 9.



Soft Link	Soft Link	Hard Link
Name = "D"	Name = "C"	Name = "A"
Value = "E"	Value = "/A/B"	Value=<address of group>

Figure 10. Structural metadata of the link objects in Figure 9.

6.4.2 Link structural metadata

Link structural metadata is described in Appendix A, *HDF5 File Format Specification*, in section II, "Level 1- File Infrastructure, Level 1A, 1B." The following structural metadata can be defined by an application program:

Name	*Description*
Link type	Link type may be “hard” or “soft.”
Name	An ASCII string.
Value	In the case of hard links, this is the address of the target object. In the case of soft links, this is an ASCII string describing a path.

6.4.3 Operations on a link

The following operations can be performed on a link object:

Name	*Description*
Insert	Inserts a link into a group.
Remove	Removes a link from a group.

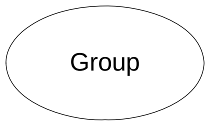
When an object is created in a group, a hard link is inserted into the group’s structural metadata.

An insert operation on a hard link creates an additional path to an already existing object. An insert operation on a soft link may result in a dangling edge in a file graph structure if the value of a soft link is not a valid path.

A remove operation on a hard link deletes one of the possible paths to an object in a file graph structure and may result in a deleted node (deleted object) and all its children, while a remove operation on a soft link keeps the number of graph nodes intact.

6.5 Group Object

6.5.1 The HDF5 group



A group object is a container within a file for zero or more HDF5 objects. Each object is a member of at least one group in a file, with the exception of the root group, which need not be a member of any group. An HDF5 object may be included in one or more groups.

Group membership is implemented via the link object. A hard link is added to a group every time an HDF5 object is created. A hard or soft link can also be added to a group to include an already existing object or an object that has not yet been created. When a hard link is removed from a group, the corresponding object pointed to by the link becomes inaccessible within that group.

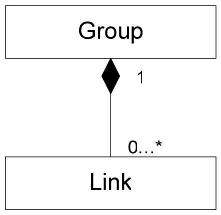


Figure 11. A link is associated with exactly one group.

6.5.2 Group structural metadata

Group structural metadata is described in Appendix A, *HDF5 File Format Specification*, in section II, “Level 1- File Infrastructure, Level 1A, 1B.”

The following structural metadata can be defined by an application program:

Name	***Description***
Data segment size of local heap	Total amount of space allocated for local heap data

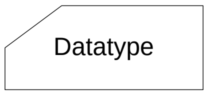
6.5.3 Operations on a group

The following operations can be performed on a group:

Name	***Description***
Create	Creates a group with defined properties.
Open	Opens an existing group.
Close	Closes a group.
Iterate	Iterates through group members.
Get information about members	Gets the number of members, a member’s object type, and the name of a link that points to a member.

6.6 Datatype Object

6.6.1 The HDF5 datatype and types of datatype objects



A datatype object describes an individual data element of a dataset or an attribute. A datatype specifies the set of possible values an element can have, the set of operations that can be performed on it, and how the values are stored.

The structural metadata of a datatype object defines the layout of a single data element and provides information required for converting the datatype to and from other datatypes of the same class.

The HDF5 data model defines two types of datatype object: named datatype (primary object) and private datatype.

A named datatype is stored in a file as a separate object and always has a hard link associated with it. A named datatype can be shared by multiple attributes and/or datasets.

A private datatype defines the datatype of a single dataset or attribute. When a dataset or attribute object is created, a private datatype's structural metadata becomes part of the dataset's or attribute's structural metadata. It can be accessed only by accessing the structural metadata of that object.

Datatypes fall into two categories: atomic and composite. An *atomic* datatype is indivisible during I/O. That is, atomic datatypes describe the smallest possible raw data element of a dataset or attribute on which I/O operations can be performed.

A *composite* datatype is one whose parts can be accessed independently during I/O. The HDF5 data model currently defines only one composite datatype – a compound datatype. A *compound* type is made up of one or more members, each of which can be accessed independently. Compound datatypes are similar to C structures or Fortran 95 derived datatypes.

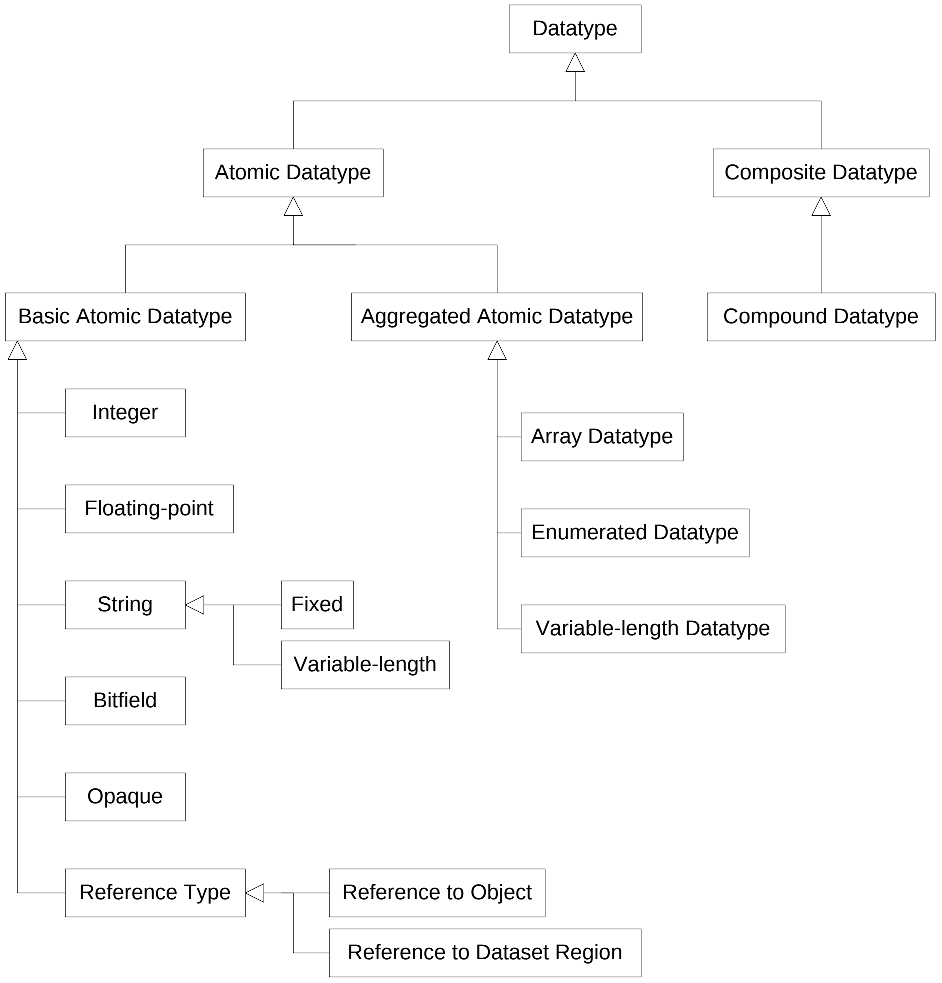


Figure 12. The hierarchy of atomic and composite HDF5 datatypes.

A member of a compound datatype can be any atomic datatype or a compound datatype. There is no restriction on the number of members or the depth of the structure.

For convenience of applications, HDF5 provides pre-defined datatypes that correspond to frequently used types such as integers and floats. The HDF5 library defines complete properties of those datatypes. The properties cannot be changed by an application. An application uses pre-defined datatypes as building blocks for atomic and composite datatypes or as parts of dataset and/or attribute definitions. Pre-defined datatypes of each class are described in the section 6.6.4, “Atomic datatypes.”

Atomic datatypes are created from pre-defined datatypes by copying and/or modifying them. Composite datatypes are created using atomic or composite datatypes.

6.6.2 Datatype structural metadata

Datatype structural metadata defines properties of a datatype. Datatype structural metadata depends on whether a datatype is named or private, and whether it is atomic or composite. These properties depend on the type of datatype and are discussed in the corresponding subsections of section 6.6.4, "Atomic datatypes."

The structural metadata of a datatype object is described in Appendix A, *HDF5 File Format Specification,* in section IV, "Disk Format Level 2- Data Objects."

6.6.3 Common operations on a datatype object

Datatype operations depend on whether a datatype is named or private, and whether it is atomic or composite. The following operations can be performed on a datatype object. Operations marked with an asterisk (*) are not allowed on a pre-defined atomic datatype:

Name	***Description***
Create*	Creates a datatype of a particular class.
Open*	Opens a named datatype.
Copy	Copies a datatype.
Close*	Closes a datatype.
Commit	Stores a datatype in a file.
Query if committed	Determines whether a datatype is stored in a file.
Compare	Determines whether two datatypes are the same.
Get/set size	Get/sets the size of a datatype.
Get class	Finds a class to which a datatype belongs.

6.6.4 Atomic datatypes

Atomic datatypes describe the smallest possible raw data element of a dataset or attribute on which I/O operations can be performed. The HDF5 data model defines a rich collection of atomic datatypes that includes integer, floating-point, string, enumeration, bitfield, array, reference, opaque, and variable-length datatypes.

It should be noted that not all datatypes are available in every programming language that the HDF5 library supports. For example, some Fortran compilers cannot read or write 1-byte or 8-byte integers. Fortran also doesn't support unsigned integers.

The following sections discuss the two broad categories of atomic datatypes: basic and aggregated atomic datatypes.

6.6.4.1 Basic atomic datatypes

Basic atomic datatypes can be used independently to define aggregated atomic datatypes and compound datatypes. Basic atomic datatypes include integer, floating-point, string, opaque, bitfield, and reference datatypes.

6.6.4.1.1 Integer datatype

An integer datatype describes an integer format and is created by copying a pre-defined integer type and setting the size, sign, and order (endianness) properties.

Operations on an integer datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Set/get sign	Defines whether the integer is signed.
Set/get order	Defines byte order in which the integer is stored.
Set/get precision	Identifies the number of significant bits.
Set/get offset	Identifies the location of significant bits.
Conversion	Converts between different types of integer numbers.

Pre-defined integer datatypes describe the most commonly used integer formats. There are two types of pre-defined integer datatypes: standard and native.

The description of a pre-defined standard integer datatype is computer system independent and is used in the definitions of the objects in the file.

The description of a pre-defined native integer datatype depends on the computer system and the computational environment (compiler settings). These datatypes are provided for the convenience of application programs to avoid requiring the discovery of the integer type properties needed to describe the integer datatype in a file. Each pre-defined native integer datatype has a corresponding pre-defined standard integer datatype that is used by HDF5 for such descriptions.

Pre-defined standard integer datatypes have names associated with them of the form H5T_STD_<base>, where <base> is a string containing the following:

- o A letter “I” for signed integers or a letter “U” for unsigned integers
- o One of the numbers 8,16,32, or 64 corresponding to the number of bits representing the integer type

- One of the strings "LE" or "BE" to indicate little-endian or big-endian byte order, respectively

Examples:

H5T_STD_I32LE represents 32-bit, little-endian, signed two's complement integer.

H5T_STD_U64BE represents 64-bit, big-endian, unsigned integer.

Pre-defined native integer datatypes have names associated with them of the form H5T_NATIVE_<language_type>, where <language_type> is a string that corresponds to a computer language-specific datatype definition.

Examples:

H5T_NATIVE_INT corresponds to the C "int" datatype.

H5T_NATIVE_ULLONG corresponds to the C "unsigned long long" datatype.

H5T_NATIVE_INTEGER corresponds to the Fortran "INTEGER" datatype.

For a complete list of pre-defined integer datatypes, see the "Predefined Datatypes" section of Appendix B, *HDF5 Reference Manual*.

6.6.4.1.2 Floating-point datatype

A floating-point datatype describes the floating-point type formats that satisfy the following conditions:

- Bits of the exponent are contiguous and stored as biased positive number.
- Bits of the mantissa are contiguous and stored as a positive magnitude.
- A sign bit exists which is set for negative values.

A floating-point datatype is created by copying a pre-defined atomic floating-point datatype and modifying its properties.

Operations on a floating-point datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Set/get sign, exponent, and mantissa	Defines the position of the sign, exponent and mantissa, and their sizes.
Set/get bias	Defines exponent bias.
Set/get normalization methods for mantissa	Determines the normalization method of the mantissa.

Name	***Description***
Set/get padding	Identifies padding for unused bits in a datum.
Set/get order	Defines byte order in which a floating-point number is stored.
Conversion	Converts between different types of floating-point numbers.

Pre-defined floating-point datatypes describe most commonly used floating-point type formats. There are two types of pre-defined floating-point datatypes: standard and native.

The description of a pre-defined standard floating-point datatype is computer system independent and is used in the definitions of objects in the file.

The description of a pre-defined native floating-point datatype depends on the computer system and the computational environment (compiler settings). It is provided for the convenience of application program to avoid requiring the discovery of the floating-point type properties needed to describe a floating-point datatype in a file. Each pre-defined native floating-point datatype has a corresponding pre-defined standard floating-point datatype that is used by HDF5 for such descriptions.

Pre-defined standard floating-point datatypes have names associated with them of the form H5T_IEEE_<base>, where <base> is a string containing the following:

- The letter “F” indicating floating-point datatype
- One of the numbers 32 or 64 corresponding to the number of bits representing the floating-point datatype
- One of the strings “LE” or “BE” to indicate little-endian or big-endian byte order, respectively

Examples:

H5T_IEEE_F32LE represents 32-bit, little-endian, IEEE floating-point.

H5T_IEEE_F64BE represents 64-bit, big-endian, IEEE floating-point.

Pre-defined native floating-point datatypes have names associated with them of the form H5T_NATIVE_<language_type>, where <language_type> is a string that corresponds to a computer language specific datatype definition.

Examples:

H5T_NATIVE_FLOAT corresponds to the C “float” datatype.

H5T_NATIVE_DOUBLE corresponds to the C “double” datatype.

H5T_NATIVE_REAL corresponds to the Fortran 95 “REAL” datatype.

For a complete list of pre-defined floating-point datatypes, see the "Predefined Datatypes" section of Appendix B, *HDF5 Reference Manual.*

6.6.4.1.3 String datatype

The HDF5 data model defines a string datatype object as a sequence of characters that is interpreted as ASCII text. This datatype is created by copying a pre-defined string datatype object and setting its size to a fixed length or to a variable length.

Because string-handling can be compiler-dependent, and to simplify programming involving strings, HDF5 defines two types of string datatypes, a null-terminated string, and a space-padded string. The pre-defined string H5T_C_S1 is stored in the same way as a C null terminated string, and the pre-defined string H5T_FORTARN_S1 is stored in the same way as a Fortran 95 space padded string.

Operations on a string datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Set/get type of string storage	Sets/gets a method of string storage to accommodate language specific storage options such as padding and termination character.

6.6.4.1.4 Bitfield datatype

The HDF5 data model defines the bitfield datatype as a sequence of bits packed in one of the integer datatypes. A bitfield datatype is created by copying a pre-defined bitfield datatype and setting its precision, offset, and padding.

Pre-defined bitfield datatypes have a name associated with them of the form H5T_<arch>_B<8,16,32,64>, where <arch> is "NATIVE" or "STD" string.

Example:

- o H5T_NATIVE_B64 – native 8-byte bit field
- o H5T_STD_B16 – standard 2-byte bit field.

Operations on a bitfield datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Set/get padding	Identifies padding for unused bits in a datum.
Set/get precision	Identifies the number of significant bits.

Set/get offset	Identifies the location of significant bits.
Conversion	Converts one bitfield datatype to another bitfield datatype.

6.6.4.1.5 Opaque datatype

The HDF5 data model provides an opaque datatype for describing data that cannot be otherwise described by HDF5. An element of an opaque datatype is treated as a blob and is not interpreted by the HDF5 library.

An opaque datatype is identified by its size and a tag, an ASCII string.

Operations on the opaque datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Set/get tag	Defines the tag, an ASCII string, for opaque datatype identification.

6.6.4.1.6 Reference datatype

The HDF5 data model defines two types of HDF5 pointers called an object reference and a dataset region reference.

6.6.4.1.6.1 Object reference datatype

A data element of the object reference datatype points to an HDF5 object.

An object reference datatype is created by copying a pre-defined object reference datatype H5T_STD_REF_OBJ.

Operations on an object reference datatype include all operations listed in 6.6.3 except the “set size” operation.

6.6.4.1.6.2 Dataset region reference datatype

A data element with a dataset region reference datatype points to a selected region of a dataset.

A dataset region reference datatype is created by copying a pre-defined dataset region reference datatype H5T_STD_REF_DSETREG.

Operations on a dataset region reference datatype include all operations listed in 6.6.3 except the “set size” operation.

6.6.4.2 Aggregated atomic datatypes

Aggregated atomic datatypes are built from any atomic or composite datatypes.

A data element with an aggregated atomic datatype is treated as one unit during I/O operations.

The HDF5 data model defines three aggregated atomic datatypes: variable-length datatype, enumeration datatype and array datatype. These datatypes are described in the following sections.

6.6.4.2.1 Variable-length datatype

The HDF5 data model defines a datatype for describing data elements that are variable-length one-dimensional arrays of some base datatype element. The size of the arrays may differ from element to element. The datatype of a base element can be any HDF5 atomic or composite datatype, including a variable-length datatype.

Variable length datatypes can be very useful for representing data records in consist of multiples of a single type. Figure 13 illustrates a one-dimensional array with a variable-length datatype, sometimes referred to as a “ragged array.” Since variable length datatypes are atomic, each element of this array must be read in its entirety. Thus, for example, the third element has five components, but it is not possible to read only one of those components in an I/O operation – all file components of the third element are read at once.

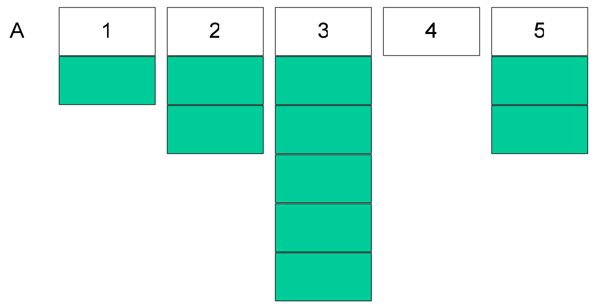


Figure 13. Each data element of a dataset with a variable-length datatype can be of a different size.

Operations on a variable length datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Get datatype of base element	Discovers the datatype of a base element.

6.6.4.2.2 Enumerated datatype

The HDF5 data model dendfines an enumeration datatype as a one-to-one mapping between a set of symbols (ASCII character strings) and a set of values, which have an integer datatype, i.e. as a set of (name, value) pairs.

Since an enumerated datatype is derived from an integer datatype, operations on an enumerated datatype include:

Name	***Description***
See section 6.6.4.1.1	All operations described in 6.6.4.1.1 can be performed.
Get number of mapping pairs	Gets the number of (name, value) pairs in an enumerated datatype.
Get value of a name	Gets the numeric value corresponding to a name.
Get name for a value	Gets the name corresponding to a numeric value.
Conversion	Converts one enumeration datatype to another enumerated datatype.

6.6.4.2.3 Array datatype

The HDF5 data model provides an array datatype for describing elements that are fixed-size multi-dimensional arrays of the same base datatype. The base datatype may be any HDF5 datatype.

Array datatypes, like strings and variable length datatypes, are included in HDF5 to provide a way to represent elements that are conceptually atomic multidimensional arrays, such as tensors.

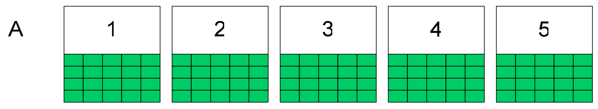


Figure 14. In an HDF5 dataset with an array datatype, each data element is itself an array.

The following operations are allowed on an array datatype:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Get datatype of base element	Discovers the datatype of a base element.
Get number of dimensions	Gets the number of array dimensions.
Get sizes of dimensions	Gets sizes of array dimensions.
Conversion	Converts an array datatype to another array datatype, if the datatype of base element allows conversion.

6.6.5 Composite datatype

The HDF5 data model currently defines only one type of composite datatype, a compound datatype.

6.6.5.1 Compound datatype

A compound datatype is a datatype that combines one or more datatypes, called members, into a more complex datatype. A compound datatype is similar to a C structure or a Fortran derived datatype. The datatype of each member can be of any HDF5 datatype, including a compound datatype. Each member of a compound datatype has a name (an ASCII string) and a byte offset indicating its position within the compound datatype.

A compound datatype is created by defining the total size of the datatype and then adding its members.

I/O operations can be performed on entire data elements of a compound datatype or on selected members of each element of a compound datatype. This characteristic distinguishes compound datatypes from atomic datatypes.

Operations on a compound datatype include:

Name	***Description***
See section 6.6.3	All operations described in 6.6.3 can be performed.
Get number of members	Gets the number of members of a compound datatype.
Get member’s info	Gets the offset, datatype, and name of a compound datatype member.
Conversion	Converts a compound datatype to another compound datatype, if the datatypes of members allows conversion.

6.7 Dataspace Object

6.7.1 The HDF5 dataspace

A dataspace object describes the dimensionality (rank) and dimension sizes of the data array associated with a dataset or attribute object.

An application program uses dataspace object for two purposes: (a) to create datasets and/or attributes and (b) to define the elements that participate in I/O operations on dataset when data is moved between the application program’s memory and a file. For the purpose of this document, only (a) will be considered.

6.7.2 Dataspace structural metadata

The structural metadata for a dataspace is described in Appendix A, *HDF5 File Format Specification*, in section IV, "Disk Format Level 2- Data Objects."

The following structural metadata is provided by an application program for a dataspace:

Name	***Description***
Rank N	Number of dimensions of a dataset
Sizes of dimensions	Current and maximum sizes for each dimension [3]

6.7.3 Operations on a dataspace

The following operations can be performed on a dataspace:

Name	***Description***
Create	Creates a dataspace.
Set dimensions	Sets sizes of current and maximum dimensions.
Close	Close a dataspace.
Copy	Creates a copy of a dataspace.
Get rank and dimensions	Gets dataspace's rank and sizes of current and maximum dimensions.
Set/get selection	Defines points in a dataspace for partial I/O (see section 8.3.1).

[3] Sizes of dimensions have to be specified only if rank is greater than 0. Dataspace with rank 0 is called a scalar dataspace. HDF5 Library provides a special API for creating such dataspaces. Only the value of rank (0) will be stored in the structural metadata for the scalar dataspaces.

6.8 Dataset Object

6.8.1 The HDF5 dataset

The HDF5 data model defines a dataset as a multidimensional array of elements along with the required structural metadata, including the description of the element datatype, spatial information about the array (the dataspace) and storage properties (how actual raw data is stored in the file).

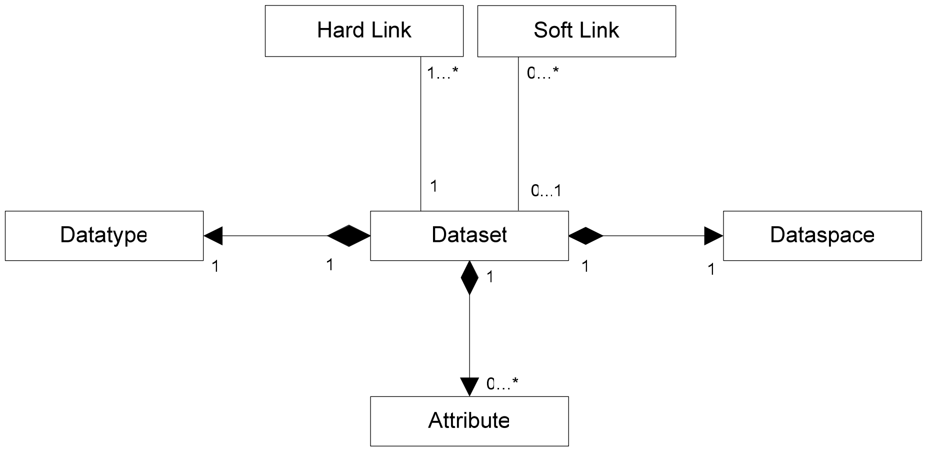


Figure 15. A dataset has a datatype, a dataspace, and optional attributes, and can be linked into the file hierarchy with both hard and soft links.

6.8.2 Dataset structural metadata

Dataset structural metadata is described in Appendix A, *HDF5 File Format Specification*, in section IV, "Disk Format Level 2- Data Objects."

Structural metadata defined by an application program includes:

Name	***Description***
Dataspace	Spatial information about a dataset
Datatype	Datatype of a dataset element
Fill value	Value returned to an application program for uninitialized data

Data storage layout o Compact o Contiguous o Chunked o External	Specification of how raw data is stored in a file In an object header As a contiguous blob In fixed-size tiles of the same dimensionality as the dataset In an external file
Filter pipeline	Description of transformations to be applied to the data stream during I/O operations
User-defined metadata	Attributes attached to a dataset

6.8.3 User-defined metadata (attribute)

A dataset may have application-defined metadata called an attribute. An attribute is a part of the dataset structural metadata. For more information on the attribute object, see section 6.9.

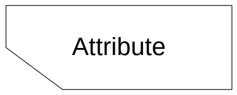
6.8.4 Operations on a dataset

Operations on a dataset include:

Name	*Description*
Create	Creates a dataset with defined properties.
Open	Opens an existing dataset.
Close	Closes a dataset.
Iterate	Iterates over elements of a dataset.
Read/Write	Reads/writes raw data to a file.
Extend	Increases current dimensions' sizes.
Query structural metadata	Gets the datatype, dataspace, storage layout and filter pipeline.
Fill with fill value	Fills uninitialized data with a fill value.

6.9 Attribute Object

6.9.1 The HDF5 attribute



The HDF5 data model defines an attribute as application-defined metadata. It comes in the form of name-value pairs, where the name is an ASCII string and the value is a scalar or a multi-dimensional array of elements.

An attribute is part of a group, dataset or named datatype definition.

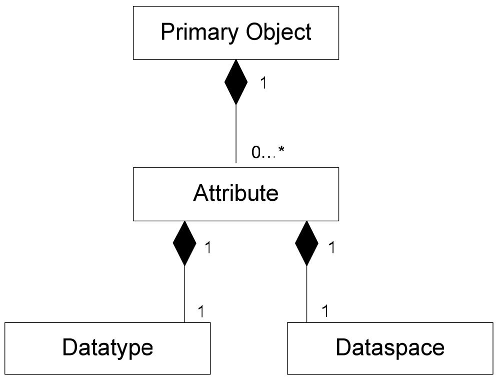


Figure 16. An attribute is a part an HDF5 primary object.

An attribute object is similar to a dataset object in that it must have an associated datatype and dataspace.

6.9.2 Attribute structural metadata

Attribute structural metadata is described in Appendix A, *HDF5 File Format Specification,* in section IV, “Disk Format Level 2- Data Objects.”

Attribute structural metadata defined by an application program includes:

Name	*Description*
Name	Attribute’s name (an ASCII string)
Datatype	Datatype of attribute’s data element
Dataspace	Dataspace of attribute’s data

6.9.3 Operations on an attribute

The following operations are defined on an attribute:

Name	*Description*
Create	Creates an attribute with defined properties and attaches it to an object.

Name	***Description***
Open	Opens an existing attribute.
Close	Closes an attribute.
Iterate	Iterates over the attributes of an object.
Delete	Deletes an attribute.
Query structural metadata	Finds the name, datatype, and dataspace of an attribute.
Read/write	Reads/writes raw data to a file.

7 HDF5 File Format

This section provides a brief introduction to the HDF5 file format. The complete description can be found in Appendix A, *HDF5 File Format Specification.*

7.1 HDF5 File Format Internal Structures

The HDF5 file format defines the low-level objects in terms of a sequence of bytes. The HDF5 objects are described in terms of the low-level objects, thus creating a mapping from the HDF5 data model to a set of byte sequences (referred to as *HDF5 logical space*).

For each HDF5 file, the *actual* layout of the byte sequences on the storage media depends on the file driver used to create the file. In other words, the HDF5 file format does not define how the HDF5 logical space is mapped to physical storage; this is done by the HDF5 library, which is described in section 8.

The table below defines the low-level objects. For a full description of the byte sequences corresponding to each object, see Appendix A, *HDF5 File Format Specification*; section references in the following table are to that document.

Name	***Description***
Super block	Contains the structural metadata about the file; for byte sequence, see section II.
B-tree node	Implements B-link trees used to store objects that can grow; for byte sequence, see section III.
Global Heap	Stores information about an object that cannot be stored in the fixed size object header message (see below). The information is usually shared between several objects in the file; for byte sequence, see section III.
Local Heap	Stores information about an object that cannot be stored in the object header fixed size message (see below); for byte sequence, see section III.
Object header	Contains information needed to identify an object and to interpret its raw data and metadata. Each object header is a set of object header messages; for the byte sequence corresponding to each header message, see section IV.

7.2 Mapping between HDF5 Objects and File Format Low-level Objects

Sections III and IV of the Appendix A, *HDF5 File Format Specification*, describe how HDF5 groups, datasets and datatypes are represented by B-trees, global and local heaps, and object headers.

A group object is represented by an object header containing a message that points to a local heap and to a B-tree that points to the object headers of the group members.

A dataset is represented by an object header containing messages that describe the datatype, dataspace, layout and other structural metadata, and a message that points to either a raw data chunk or to a B-tree that points to the raw data chunks.

A datatype is represented by an object header containing a datatype message.

While object headers, heaps, and B-tree nodes are represented by contiguous byte sequences, the complete description of an object may not be a contiguous set of byte sequences. Therefore, in general, there is no way to discover an object in an HDF5 file by specifying a file offset and a size of the object; a special library is needed.

Section 8 describes the HDF5 library and the set of application programming interfaces that provide access to an HDF5 file and to the objects stored in the file.

8 HDF5 I/O Library

The HDF5 library implements the HDF5 data model and storage model according to the specifications described in sections 6 and 7 of this document and in Appendix A, *HDF5 File Format Specification*. The following sections introduce the HDF5 programming model and the HDF5 application programming interfaces (APIs). For a complete set of the HDF5 APIs, see Appendix B, *HDF5 Reference Manual*.

8.1 Introduction to the HDF5 Programming Model

The HDF5 programming model defines how operations are performed on HDF5 objects. It consists of five steps, as described in the table below. Steps marked with an asterisk (*) are optional.

Step	***Description***	***Example***
1*	Properties of an HDF5 object are defined.	Properties are set to create an extendible dataset in step 2.
2	The HDF5 object is accessed (opened or created).	Dataset is created; operations can be performed on the dataset.
3*	Properties of operations on the HDF5 objects are defined.	Size of type conversion buffer is set to tune performance.
4	Operations on the HDF5 object are performed.	Dataset is written to the file.
5	Access to the HDF5 object is terminated.	Access to the dataset is terminated; any further operation on it will fail.

The order of the steps is important. Step 1 has to be done before steps 2 – 5 (i.e. immutable structural metadata has to be defined before an object is created and accessed); obviously, step 3 has to be performed before step 4; step 5 can be performed only if step 2 is complete; step 4 cannot be performed if access to an object has been terminated (step 5).

The HDF5 library refers to an object via an object identifier. An identifier is created when the object is opened or created. The identifier is used to invoke subsequent operations on the object. The identifier is also used to specify dependencies between objects. For example, a dataset is always created under some group; the creation operation on the dataset therefore can use the group identifier to specify the location of the dataset in the file hierarchy.

8.2 HDF5 APIs

The HDF5 APIs enforce the HDF5 data model and the HDF5 programming model. An application program uses HDF5 APIs to map its data structures to HDF5 objects and to perform operations on those objects.

The complete set of the HDF5 APIs is implemented in the C language; a subset of the APIs is also implemented in the Fortran 90, C++ and Java languages. This proposal describes the HDF5 C APIs only.

The names of the C HDF5 APIs follow the following convention: H5<A,D,E,F,G,I,P,R,S,T,Z><string>, where <string> is a sequence of lower case characters "a – z" and "_" and always starts with an alphabetic character. Each of the capital letters corresponds to a set of APIs that implements operations on a particular HDF5 object or on an HDF5 library object. The table below summarizes the HDF5 APIs.

Prefix	***Description (examples)***
H5A	Operations on attribute objects (H5Acreate, H5Awrite, H5Aclose)
H5D	Operations on dataset objects (H5Dopen, H5Diterate)
H5E	Error handling operations (H5Eprint, H5Eclear)
H5F	Operations on file objects (H5Fis_hdf5, H5Fmount)
H5G	Operations on group objects (H5Gcreate, H5Gunlink)
H5I	Operations on object identifiers (H5Iget_name, H5Iget_type)
H5P	Operations to set and modify object properties and properties of operations on objects (H5Pcreate, H5Pset_fapl_family, H5Pset_chunk)
H5R	Operations on references to objects and references to dataset regions (H5Rcreate, H5Rdereference, H5Rget_obj_type)
H5S	Operations on dataspace objects (H5Screate_simple, H5Sselect_hyperslab)
H5T	Operations on datatype objects (H5Tcopy, H5Tget_size)
H5Z	Operations on user-defined filters

Appendix B, *HDF5 Reference Manual*, includes lists of all HDF5 C and Fortran 90 APIs and complete descriptions of each.

8.3 The HDF5 Library I/O Features

The following sections introduce three main features of the HDF5 I/O library: operations to perform partial I/O on raw data, data transformation operations on each element of the raw data during I/O operations (HDF5 filters), and special-purpose I/O mechanisms to specify physical storage layouts (virtual file layer or VFL).

8.3.1 Partial I/O

The HDF5 library provides a mechanism called *hyperslab selection* for selecting elements of a data array that participate in I/O operations (partial I/O).

HDF5 defines two types of basic selections: point selection and simple hyperslab selection. All selections are built from one of those two types using set operations:

- union
- difference
- intersection
- complement

Selection type	***Description***
Point selection	One element of a dataset, identified by its array indices
Simple hyperslab selection (also called a block)	A contiguous sub-array identified by an offset from the beginning of the array in each dimension, and the size in each dimension

HDF5 library provides a special API to describe commonly used selection of equally spaced and equally sized blocks of data. Such selection is characterized by the offset of the first block in the data array, sizes of the block, spaces (or strides) between blocks in each dimension and number of blocks in each dimension. The same API is used to describe a simple hyperslab selection (a contiguous sub-array).

Set operations on hyperslab or point selections allow the definition of very complex, irregularly shaped subsets of an N-dimensional array. During I/O operations, the shape and the dimensionality may not be preserved, i.e. a 2-dimensional sub-array from a dataset in a file (source) may be scattered to an irregularly shaped subset of a 3-dimensional array in memory (destination) during the read operation. The number of elements in both selections, however, must remain the same.

Elements in each selection are ordered according to the C ordering rule (row-column). The i-th element in the source selection is transformed (read/written) to the i-th element in the destination selection.

8.3.2 Filters

The HDF5 library defines transformations (filters) on each element of a dataset that participates in an I/O operation. Filters may be combined to create a filter pipeline. When an element goes through the filter pipeline, the result of one transformation becomes the input to the next transformation in the pipeline.

Permanent transformations on elements are defined when a dataset is created and cannot be removed after that. Transient transformations are defined on every I/O operation. The HDF5 library standard defines only the permanent transformations described in the table below. Transformations marked with an asterisk (*) may be performed only on the elements of chunked datasets.

Transformation type	***Description***
Datatype conversion	Converts numeric values of the same class (integer to integer, or floating-point to floating-point).
Raw data shuffling*	Rearranges bits in each element to achieve better compression ratio.
Raw data compression*	Raw data is stored in compressed form; two types of compressions are supported:[4] o GNU zlib compression[5] o NASA szip compression[6]
Raw data error detection*	Error-detecting codes (EDC) provide a way to identify data that has been corrupted during storage or transmission; HDF5 supports the Fletcher32 algorithm.

HDF5 library can be configured to enable or disable a specified filter or a set of filters. If filter is not enabled, HDF5 library will not be able to read the data if the filter was applied when data was written.

Szip compression is licensed software. Under some circumstances (commercial usage only) a license is needed to write szip-compressed data. *No License is needed for reading szip-compressed data.*

See the following link for more information:
`http://hdfgroup.com/doc_resource/SZIP/Commercial_szip.html`

[4] Both compression methods are optional.
[5] For information regarding zlib compression, see `http://www.zlib.org/`.
[6] For information regarding szip compression, see `http://hdfgroup.org/doc_resource/SZIP/`.

8.3.3 Virtual file layer (VFL)

The HDF5 file format describes how HDF5 data structures and dataset raw data are mapped to a linear format address space (HDF5 logical file). The HDF5 library implements the mapping in terms of an API. However, the HDF5 format specifications do not indicate how the format address space is mapped onto storage and how the data in the storage is accessed. The virtual file layer, or VFL, allows an application to design and implement its own mapping between the HDF5 logical file and storage. The VFL also specifies the type of access.

The proposed standard defines four different storage layouts (see 6.3.1) and corresponding access methods:

Storage layout	***Access method***
Single file on a storage media	POSIX unbuffered I/O POSIX buffered I/O MPI I/O
Multi files	POSIX unbuffered I/O
Family of files	POSIX unbuffered I/O
Core (memory) file	Memory access

9 Authors' Address

Mike Folk, Elena Pourmal, The HDF Group, 1901 So. First St, Suite C-2, Champaign, Il 61820 USA. Telephone: 217-333-0238; fax: 217-333-9049; email: mfolk@hdfgroup.org, epourmal@hdfgroup.org.

10 Appendix A – HDF5 File Format Specification

11 Appendix B – HDF5 API Reference Manual

12 Appendix C – Glossary of acronyms

Acronym	Description
HDF5	Hierarchical Data Format, version 5
CCSDS:	Consultative Committee for Space Data Systems
ECS:	EOSDIS Core System
EOSDIS:	Earth Observing System Data and Information System

13 Appendix D – Errata

There are no errata for the document.

14 Appendix E – HDF5 copyright