HDDM User's Guide

R.T. Jones
University of Connecticut
(Dated: February 9, 2021)

The HDDM (Hierarchical Document Data Model) is an xml schema for expressing the meaning and relationships of streaming data from scientific instruments. The design is based on a hierarchical network where each node in the graph has a single parent node, multiple key-value attributes, and an arbitrary number of child nodes, similar to a the elements in an xml document. The model is adapted specifically to the case of repetitive data models such as appear in the data stream from a high-energy physics experiment. HDDM is designed to support the evolution of a data model over time, such that the same binary can read streams generated with previous versions of the model, and old binaries can read streams generated with more recent versions of the model, subject to very general constraints on model evolution. Conceptual representation of the data as an xml document is an essential design feature, but instantiation in memory does not involve the creation of explicit textual elements or construction of an explicit Document Object Model (DOM) for the data. The HDDM toolkit includes tools to express HDDM streams in xml, check their validity against the schema, and serialize/deserialize from container objects in memory. Originally written in c, HDDM provides application programmer interfaces for C++ and python as well. In addition to its own native data format, applications that use HDDM to access their data can also read/write standard HDF5 files and ROOT trees.

HDDM release 1.4

Contents

```
<HDDM class="x" version="1.0" xmlns="http://www.gluex.org/hddm">
  <student minOccurs="0" name="string">
    <enrolled maxOccurs="unbounded" semester="int" year="int">
      <course credits="int" maxOccurs="unbounded" title="string">
        <result Pass="boolean" grade="string" />
      </course>
    </enrolled>
  </student>
</HDDM>
```

FIG. 1: A simple example of a HDDM data model template. A HDDM data stream encodes a sequence of `<student>` records, each with the same internal hierarchical structure defined in the template. The data values designated by simple types, "int", "string", etc. are packed sequentially into the byte stream on output, and unpacked into memory-resident objects on input.

I. INTRODUCTION

The HDDM toolkit provides the scientist with a means to format streaming data from a scientific instrument into a structured self-describing byte stream of binary data that is platform-independent and easy to browse, filter, transform extend, annotate, and validate using standard xml tools. The purpose of this User's Guide is to describe the use and operation of the HDDM tools, describing how to define a new data model or inspect an existing one, how to create new HDDM files or read data from existing files. HDDM tools automatically generate the object classes that represent the data described in the user's model, with methods to access the object data in memory as well as to serialize/deserialize themselves between memory and an external byte stream connected to an ordinary file on disk or to a network socket. The underlying implementation of the i/o library is in C++, so it provides good performance in terms of data rate to/from byte streams, with optional on-the-fly compression/decompression and data integrity verification. In addition to ordinary sequential access to data, random-access to records at an arbitrary offset in a stream is also provided for streams that support random seeks, without the need to read the entire stream.

II. XML SCHEMAS AND HDDM TEMPLATES

Every HDDM stream has an associated data model, expressed either in the form of a standard XML schema, or more compactly, in the form of a HDDM template. Tools are provided to translate between the schema and HDDM template description of the model. A HDDM template is a short xml document that describes the structure of one record in the HDDM stream. Every HDDM stream has a copy of its template in plain-text UTF-8 at the beginning, followed by a sequence of data records in a compact binary format. The template contains all of the information necessary to reconstruct the original data objects from the serialized records, together with their hierarchical arrangement. A simple example of a template is given in Fig. 1.

All of the records in the file represent repeats of this basic structure, with different values in the data fields. All actual data values are represented as attributes of tags. Attributes that are assigned type names ("string", "int", "long", "float", "double", "boolean", "anyURI", and "Particle_t") are user data. Any other values assigned to attributes other than these simple types are treated as annotations in the data model, eg. to specify the units assigned to physical values, and do not take up space in the file (other than in the template header). Some of these literal attributes function as metadata, eg. you might want to add an attribute unit="GeV" to document the units used for other attributes in a tag. Other special attributes like minOccurs/maxOccurs take special values that tell the data model whether a given element is always present in every record or may be omitted (minOccurs="0") or whether it may be repeated any number of times (maxOccurs="unbounded"), as in a standard xml schema. The top-level element is special in that it must always be named HDDM and have the attributes shown in Fig. 1. The class attribute of the `<HDDM>` element is any (preferably short) string that you choose for the family of data models you are creating. Choose a short, unique name for your class because it is used in the type names of user objects that are defined written by the HDDM user library. Its purpose is to prevents collisions between different HDDM stream types that may coexist in a single application.

Templates provide an intuitive way of specifying the structure of a data record in a HDDM stream. For most users, this is all they need to know about in order to define their data models. For those familiar with XML schema validation, there is a more formal way to specify the structure of an xml document

which is called a *xml schema*. HDDM uses schemas in two different ways. The first is to specify the structure of the templates themselves. The template shown in Fig. 1 conforms to a schema called `http://www.gluex.org/hddm`. This is not a URL to anywhere; it is a URI known as an *xml namespace*, as suggested by the name of the `xmlns` attribute in the HDDM tag of the template. The schema for this document type is found in hddm_schema.xsl in the schema directory of the distribution. The second use of schemas is related to the fact that every record in a HDDM stream is a valid xml fragment corresponding to a schema against which it can be validated. The HDDM toolkit provides a pair of tools *hddm-schema* and *schema-hddm* that convert back and forth between templates and schema. The two are equivalent ways of representing the same information about the structure of a HDDM record, with the schema being more complete and standards-based, while the template is shorter and more intuitive to most users. Schemas provide a much more general set of constraints that can be expressed for the data and relationships between them, but experience has shown that their practical use for this purpose is limited to special instances where standards-based data validation must be performed. The remainder of this document deals mainly only with templates.

A. rules for constructing HDDM templates

1. A hddm template is nothing more than a plain-text xml file that mimics the structure of the xml that the program expects on input or produces on output.
2. The top element in the template must be <HDDM> and have three required attributes: *class*, *version*, and *xmlns*. The value of the latter must be `xmlns="http://www.gluex.org/hddm"`. The values of the class and version arguments are user-defined. They serve to identify a family of schemas that share a common set of elements. See below for more details on classes.
3. The names of elements below the root `<HDDM>` element are user-defined, but they must be constructed according to the following rules.
4. All values in HDDM files are expressed as attributes of elements. Any text that appears between tags in the template is treated as a comment.
5. An element may have two kinds information attached to it: child elements which appear as new tags enclosed between the open and close tags of the parent element, and attributes which appear as `key="value"` items inside the open tag.
6. All variable quantities in the data model are carried by named attributes of elements. The rest of the document exists to express the meaning of the data and the relationships between them.
7. All elements in the model document either hold attributes, contain other elements, or both. Empty elements are meaningless, and are not allowed.
8. A template contains type names in the place of actual numerical or string values for the fields in the structure. For example, instead of showing `energy="12.5"` as might be shown for sample data, the template would show in this position `energy="float"` or `energy="double"`.
9. The complete list of allowed types supported by HDDM is “int”, “long”, “float”, “double”, “boolean”, “string”, “anyURI”, and “Particle_t”. The Particle_t type is a value from an enumerated list of capitalized names of physical particles. The `int` type is a 32-bit signed integer, and `long` is a 64-bit signed integer. The meaning of the other simple types should be clear from the names.
10. Attributes in the template that do not belong to this list are treated as constants which annotating the xml record. The must have the same value each time a given element is repeated throughout the template.
11. Any given attribute may appear more than once throughout the template hierarchy. Wherever it appears, it must appear with identical attributes and with content elements of the same order and type.
12. Another difference between a template sample data is that the template never shows a given element more than once in a given context, even if the given tag would normally the repeated more than once for an actual sample. One obvious example of this is a main record element, which is represented only once in the template, but repeated multiple times in a HDDM stream.

13. By default, it is assumed that an element appearing in the template must appear in that position exactly once. If the element is allowed to appear more than once or not at all then additional attributes should be supplied of the form `minOccurs="N1"` and `maxOccurs="N2"` where `N1` can be zero or any positive integer and `N2` can be any integer no smaller than `N1`, or set to the string "unbounded". Each defaults to 1.

14. Arrays of simple types are represented by a sequence of elements, each carrying an attribute containing a single value from the array. This is more verbose than a simple space separated string of values would be, but it is more apt for expressing parallelism between related arrays of data.

15. An element may be used more than once in the model, but it may never appear as a descendant of itself. Such recursion is complicated to handle and it is difficult to think of a situation where it is necessary.

16. Because templates contain new tags that are invented by the programmer, it is not possible to write a standard template schema against which a new template can be validated. In the place of schema validation, one should use the `hddm-schema` tool to check a xml file for correctness as a hddm template. Any errors that occur in the hddm - schema transformation indicate problems in the template that must be fixed before it can be used to generate a HDDM library.

B. rules for constructing HDDM schemas

1. HDDM schemas must be valid xml schemas, belonging to the namespace `http://www.w3.org/2001/XMLSchema`. Not every valid xml schema is a valid HDDM schema because HDDM is much more specific in how the template should be organized than is required by the rules for well-formed xml.

2. In the following specification, a prefix `xs:` is applied to the names of elements, attributes or datatypes that belong to the official schema namespace `http://www.w3.org/2001/XMLSchema`, whose meaning is defined by the xml schema standard. The specific rules for HDDM schemas are represented by the private namespace `http://www.gluex.org/hddm` denoted by the prefix `hddm:`.

3. The top element defined by the schema must be `<HDDM>` and have three required attributes: `class`, `version`, and `xmlns`. The value of the latter must be `xmlns="http://www.gluex.org/hddm"`. The class and version arguments are of type `xs:string` and are user-defined. They serve to identify a group of schemas that share a basic set of tags.

4. The names of elements below the root `<HDDM>` element are user-defined, but they must be constructed according to the following rules.

5. An element may have two kinds of content: child elements and attributes, and hence must have schema type `xs:complexType`. Elements represent the grouping together of related pieces of data in a hierarchy of nodes. The actual numerical or symbolic values of individual variables appear as the values of attributes.

6. All quantities in the data model are carried by named attributes of elements. The rest of the document exists to express the meaning of the data and the relationships between them.

7. All elements in the model document either hold attributes, contain other elements, or both. Empty nodes are meaningless, and are not allowed.

8. Text content between open and close tags is allowed in documents (`type="mixed"`) but it is treated as a comment and stripped on translation to a template. Basic HDDM schemas do not use `type="mixed"` elements.

9. The datatype of an attribute is restricted to a subset of basic types to simplify the task of translation. Currently the list is `xs:int`, `xs:long`, `xs:float`, `xs:double`, `xs:boolean`, `xs:string`, `xs:anyURI`, and `hddm:Particle_t`. User types that are derived from the above by `xs:restriction` may also be defined and used in a HDDM schema.

10. Attributes must always be either "required" or "fixed". Default attributes, i.e. those that are sometimes present inside their host and sometimes not are not allowed. This allows a single element to be treated as a fixed-length binary object on serialization, which has advantages for efficient i/o.

11. A datum that is sometimes absent can be expressed in the model by creating a sub-element to be contained within the host element with `minOccurs="0"`, and assigning the optional value as an attribute of the sub-element.

12. Fixed attributes (with `use="fixed"`) may be attached to user-defined elements. They may be of any valid schema datatype, not just those listed above, and may be used as annotations to qualify the information contained in the element. Because they have the same value for every instance of the element, they do not take up space in the binary stream, but they are included explicitly in the output produced by the **hddm-xml** translator.

13. All elements must be globally defined in the schema, i.e. declared at the top level of the `xs:schema` element. Child elements are included in the definition of their parents through a `ref=tagname` reference. Local definitions of elements inside other elements are not allowed. This guarantees that a given element has the same meaning and contents wherever it appears in the hierarchy.

14. Arrays of simple types are represented by a sequence of elements, each carrying an attribute containing a single value from the array. This is more verbose than allowing a simple list type defined by `xs:list`, but the chosen method is more apt for expressing parallelism between related arrays of data, such as frequently occurs in descriptions of physical events. Forbidding the use of simple `xs:list` datatypes should encourage programmers to chose the better model, although of course they could just mimic the habitual use of indexed lists by filling the data tree with long strings of monads.

15. Elements are included inside their parent elements within a `xs:sequence` schema declaration. Each member of the sequence must be a reference to another element with a top-level definition.

16. A given element may occur only once in a given the sequence, but may have `minOccurs` and `maxOccurs` attributes to indicate possible absence or repetition of the element.

17. The `sequence` is the only content organizer allowed by HDDM. More complex organizers are supported by schema standards, such as `all` and `choice`, but their use would complicate the i/o interfaces that have to handle them and they add little by way of flexibility to the model the way it is currently defined.

18. An element may be used more than once in the model, but it may never appear as a descendant of itself. Such recursion is complicated to handle and it is difficult to think of a situation where it is necessary.

19. A user can check whether a given schema conforms to the HDDM rules by transforming it into a hddm template document. Any errors that occur during the transformation generate a message indicating where the specification has been violated.

C. class relationships and model evolution

1. Two HDDM schemas belong to the same class if all tags that are defined in both have the same set of attributes in both.

2. This is a fairly weak condition. It is possible that all data files used in a particular application will belong to the same class, but it is not required.

3. If two HDDM schemas belong to the same class then it is possible to form a union schema that will validate documents of either type by taking the xml union of the two schema documents and changing any sequence elements in one and not in the other to `minOccurs="0"`.

4. The translation tools **xml-hddm** and **hddm-xml** will work with any HDDM class.

5. Any program built using the i/o library created with **hddm-c** or **hddm-cpp** is dependent on the class of the schema used during the build. Any files it writes through this interface will be built on this schema. moreover it is able to read any file of the same class without recompilation.

6. A new schema may be derived from an existing HDDM schema by taking the existing one and adding new elements to the structure. In this case the version attribute of the HDDM tag should be incremented, while leaving the class attribute unchanged.

7. A program that was built against the libraries built using the `hddm-c` or `hddm-cpp` tools can read from any from any hddm file of the same class as the original schema used during the build. If the content of the file is a superset of the original schema then nothing has changed. If some elements of the original schema are missing in the file then the i/o still works transparently, but the data elements corresponding to the missing branches of the data model graph will be empty, i.e. zeroed out.

8. The c/c++ i/o library will reject an attempt to read from a hddm file that has a schema of a different class from the one for which it was built.

9. No mandatory rules are enforced on the `version` attribute of the hddm file, but it is available to programs and may be used to select certain actions based on the "vintage" of the data.

10. Programs that need simultaneous access to multiple classes of hddm files can be built with more than one i/o library. The structures and i/o interface are defined in separate header files hddm_`X`.h and implementation files hddm_`X`.c or hddm_`X`++.cpp, where `X` is the class letter.

III. OVERVIEW OF THE HDDM TOOLKIT

The HDDM toolkit is distributed as a github repository `https://github.com/rjones30/HDDM`. Instructions for how to download and build HDDM are given in the INSTALL file provided at the top level of the download tree. The HDDM tools are installed by the installation procedure into the bin directory under the installation base. Before continuing to read this document, make sure that the basic HDDM tools including hddm-xml, xml-hddm, hddm-c, hddm-cpp, hddm-py, and xml-xml are in your shell PATH. These tools are not the HDDM libraries themselves, but the code generators you need to construct user-callable libraries from your HDDM template.

If you already have a HDDM data file that you want to read, you can generate the i/o user library that you can use to read from it and optionally to write a selection of the records to a new HDDM output file. The template that the code generators need to generate the user library is present in the header of the HDDM file that you want to read. Simply providing the data file as input to `hddm-c` will generate c header and implementation files that you need to include on the compiler command line together with your c application code for your project, and similarly, `hddm-cpp` in the case of C++ applications, or `hddm-py` to generate a python module. Of the three supported programming languages, the python implementation is the least verbose and most readable, so it is recommended as a starting point for someone experimenting with HDDM.

Independent of any user programs or language-specific API, the HDDM toolkit provides two tools that can be used to read and write HDDM files directly from the command line. The following command accepts any valid HDDM file as input and prints the contents of the file in plain-text xml to standard output.

```
$ hddm-xml [-n <count>] [-o <output.xml>] <datafile.hddm> [...]
```

The reverse action is provided by the `xml-hddm` tool.

```
$ xml-hddm [-n <count>] -t template.xml <input.xml> [...]
```

The full XML rendition of a data file with many records is highly verbose, which makes the plain-text xml rendering of a HDDM stream of limited practical interest, except as a means to visually browse the data, or to make small changes using a text editor. The reversibility of the conversion between xml and HDDM representations can be useful in cases where one might doubt the fidelity of the encoding being used by HDDM. These two tools do not require any compile-and-link step each time the template is changed, so they are very useful to quickly inspect the contents of a HDDM file. Keep them handy when working through the language-specific procedures below.

IV. HDDM IN PYTHON

If you have access to a HDDM file that was already written, copy it into your work directory and use it as a template for building a python module to access the model data as python list objects. Otherwise, the HDDM package distribution directory contains a simple example in `models/exam1x.hddm` that you can use for this purpose. Copy the HDDM file you are using for this test into a new project directory, and enter the following commands to build the python module for this data model. If you encounter the error, "command not found", make sure that the bin directory where you installed the HDDM package is somewhere in your shell PATH.

```
$ hddm-cpp exam1x.hddm  # builds the underlying C++ library
$ hddm-py exam1x.hddm   # builds the python interface
$ python setup_hddm_x.py # creates the module hddm_x
```

In this example, I assigned 'x' as the HDDM *class* abbreviation (see the HDDM tag in the template header). You should change it to whatever class abbreviation you choose for your HDDM data model. The above steps should create a python module in the form of a shared library that starts with `hddm_x` in your project directory. Copy that module to a directory in your PYTHONPATH where you usually place your private python modules, or add your project directory to your PYTHONPATH.

Execute the following interactive python script to print the contents of the example HDDM file in plain text.

```
import hddm_x
for rec in hddm_x.istream("exam1x.hddm"):
   print(rec)
```

To see the same data printed out as a properly formatted xml document, replace the `print(rec)` in the above python HDDM reader with `print(rec.toXML())`. If the above command generated no output then your input HDDM file is empty, as it would be if you used the example input file `models/exam1x.hddm`. After you have written some data to an HDDM file, as explained in the next section, come back and try it again. The full set of methods and attributes supported by the python module is displayed by the command, "`pydoc hddm_x`".

A. writing HDDM files in python

For this example, let us continue using the same template as was used in the example python HDDM reader above. You should already have built and installed the hddm_x python module and installed it in your PYTHONPATH, using the build steps listed above. Execute the following python script to write a new output HDDM file from scratch, using and some test user data.

```
import hddm_x
ofs = hddm_x.ostream('exam2x.hddm')
xrec = hddm_x.HDDM()
student = xrec.addStudents()
student[0].name = 'Humphrey Gaston'
enrolled = student[0].addEnrolleds()
enrolled[0].year = 2005
enrolled[0].semester = 2
course = enrolled[0].addCourses(3)
course[0].credits = 3
course[0].title = 'Beginning Russian'
result = course[0].addResults()
result[0].grade = 'A-'
result[0].Pass = True
course[1].credits = 1
course[1].title = 'Bohemian Poetry'
result = course[1].addResults()
result[0].grade = 'C'
result[0].Pass = 1
course[2].credits = 4
course[2].title = 'Developmental Psychology'
result = course[2].addResults()
result[0].grade = 'B+'
result[0].Pass = True
ofs.write(xrec)
```

This script generates a new HDDM file called `exam2x.hddm`. Now running the 3-line python reader from the previous section on `exam2x.hddm` should yield the following output.

```
HDDM
  student name="Humphrey Gaston"
    enrolled semester=2 year=2005
      course credits=3 title="Beginning Russian"
        result Pass=false grade="A-"
      course credits=1 title="Bohemian Poetry"
        result Pass=false grade="C"
      course credits=4 title="Developmental Psychology"
        result Pass=false grade="B+"
```

The example writer above began by creating a new record by calling the `HDDM()` record constructor. Then it populated the structure top-down, calling `add`XXXs() methods for each tag XXX under that, where XXXs refers to the name of the tag element in the template transformed into its capitalized-plural form. The addXXXs() methods take a single optional `int` argument, which is the number of copies of that element that need to be added (default 1). They return a list that can be indexed in the usual python fashion to give access to the individual members of the list. Each of these has `add`XXXs() methods for each of its contents, and so on down the tree. You can omit whole branches of the tree by simply not calling the corresponding `add`XXXs() method. Xml rules require that you specify minOccurs="0" in the template for the container tag if you plan to make that subtree optional. As soon as a new element list is created, you can fill in the values of its attributes using simple assignment semantics, as illustrated in the example. The names of the python data members are the same as the names of the attributes in the template.

B. reading HDDM files in python

For this illustration, you are assumed to have created the file exam2x.hddm using the writer described in the previous section. The following python program lets you open this file and extract bits of information from the first record, writing a summary report at the end. Of course, in actual practice, a HDDM file would contain many records and the analysis would loop over many instances of student.

```
import hddm_x
ifs = hddm_x.istream("exam2x.hddm")
xrec = ifs.read()
total_enrolled = 0
total_courses = 0
total_credits = 0
total_passed = 0
for course in xrec.getCourses():
   total_courses += 1
   if course.getResult().Pass:
      if course.year > 1992:
         total_credits += course.credits
      total_passed += 1
   total_enrolled += 1
   studentname = course.name

print(studentname, "enrolled in", total_courses, " courses",
       "and passed" , total_passed, "of them,\n",
       "earning a total of", total_credits, "credits.")
```

Running the above code should produce output like the following:

```
Humphrey Gaston enrolled in 3 courses and passed 3 of them,
 earning a total of 8 credits.
```

In addition to each tag supporting the lookup (via getXXXs methods) of the tags immediately appearing under it in the template hierarchy, the top-level HDDM record provides global getXXXs methods for every tag throughout the hierarchy, and returns all instances of a given tag that appear anywhere in the record, in the order of their appearance. The istream object itself also functions as an iterable in python so the construct, `for rec in hddm_x.istream('exam2x.hddm'):` would look over all records in the input file, assigning the `rec` iteration variable to each record as it is read from the input stream. Likewise, each call to method getXXXs() returns a python list of tag element objects that is iterable using the usual python `for` semantics, as illustrated for xrec.getCourses() above. As before, the individual attributes of each tag instance are accessed as plain data members of their host object. The standard python list functions (eg. len(list), str(list), repr(list)) all work as expected for these hddm tag list objects returned by getXXXs() method. These natural python iteration and accessor semantics provide a quick-and-simple prototyping framework for analysis of repetitive experimental data.

A slightly more complex example of reading and writing HDDM streams based on this example template is found in the distribution under `examples/exam2.py`.

C. python API reference

V. HDDM IN C++

If you have access to a HDDM file that was already written, copy it into your work directory and use it as a template for building a C++ library to access the model data as C++ objects. Otherwise, the HDDM package distribution directory contains a simple example in `models/exam1x.hddm` that you can use for this purpose. The following commands build the C++ library corresponding to your HDDM model.

```
$ hddm-cpp exam1x.hddm
$ g++ -std=c++11 -c hddm_x++.cpp -I $HDDM_INSTALL_DIR/include \
  -L $HDDM_INSTALL_DIR/lib64 -lxstream -lz -lbz2
$ ar -r libhddm_x.a hddm_x++.o
```

If the environment variables in this command are not defined in your shell environment, define them or replace them with the appropriate values.

A. writing HDDM files in C++

This section turns once again to the example template `exam1x.hddm` used earlier under the python hddm writer section. Having already built a C++ library against this template, now it is time to write a user application that uses the library to create HDDM output according to the template. Open a new C++ source file in an editor and cut/paste the contents of the box below into it, then save it.

```
#include <fstream>
#include "hddm_x.hpp"
int main()
{
   // build the nodal structure for this record and fill in its values
   hddm_x::HDDM xrec;
   hddm_x::StudentList student = xrec.addStudents();
   student().setName("Humphrey Gaston");
   hddm_x::EnrolledList enrolled = student().addEnrolleds();
   enrolled().setYear(2005);
   enrolled().setSemester(2);
   hddm_x::CourseList course = enrolled().addCourses(3);
   course(0).setCredits(3);
   course(0).setTitle("Beginning Russian");
   course(0).addResults();
   course(0).getResult().setGrade("A-");
   course(0).getResult().setPass(true);
   course(1).setCredits(1);
   course(1).setTitle("Bohemian Poetry");
   course(1).addResults();
   course(1).getResult().setGrade("C");
   course(1).getResult().setPass(1);
   course(2).setCredits(4);
   course(2).setTitle("Developmental Psychology");
   course(2).addResults();
   course(2).getResult().setGrade("B+");
   course(2).getResult().setPass(true);

   std::ofstream ofs("exam2x.hddm");
   hddm_x::ostream ostr(ofs);
   ostr << xrec;
   xrec.clear();
   return 0;
}
```

Save this C++ program to a file named `write_exam.cpp` and compile it into an executable using a command like the following.

```
$ g++ -std=c++11 -o write_exam write_exam.cpp hddm_x++.o -I. -I $HDDM_INSTALL_DIR/include \
  -L $HDDM_INSTALL_DIR/lib64 -lxstream -lz -lbz2
```

The paths listed in the compilation command line may need to be customized for your own build environment. Once it completes successfully, will find the executable `write_exam` in the working directory. Run it as `./write_exam2` and it should create a new HDDM file called `exam2x.hddm`. Running `hddm-xml write_exam2x.hddm` should produce output like the following.

```
HDDM
  student name="Humphrey Gaston"
    enrolled semester=2 year=2005
      course credits=3 title="Beginning Russian"
        result Pass=false grade="A-"
      course credits=1 title="Bohemian Poetry"
        result Pass=false grade="C"
      course credits=4 title="Developmental Psychology"
        result Pass=false grade="B+"
```

The example begins by creating an empty record by calling the `HDDM()` default constructor. Then it populates the structure top-down by calling `add`XXXs() methods for each tag XXX under that, where XXXs represents the name of the tag element in the template transformed into a capitalized-plural form. The `add`XXXs() methods take a single optional `int` argument, which is the number of instances of that element to be added to the container element (default is 1). They return a subclass of `std::list` that can be iterated over in the usual fashion, or indexed with `operator()(int)` to access the individual members of the list. Each of these has `addXXXs()` methods for each of its contents, and so on down the tree. You can omit whole branches of the tree by simply not calling the corresponding `addXXXs()` method, although xml rules require that you specify minOccurs="0" for the container tag in the template if you plan to do that. As soon as a new element list is constructed, you can fill in the values of its object attributes using set<attname> methods, as illustrated in the example, where <attname> is a capitalized version of the names of the attribute in the template.

B. reading HDDM files in C++

This section assumes that you have created the file exam2x.hddm using the procedure described in the previous section. The following C++ program opens this file and extracts bits of information from the first record, and writes a summary report. Of course, in actual practice such a data file would probably contain many records, and the analysis would loop over many instances of student.

```
#include <fstream>
#include "hddm_x.hpp"
int main()
{
   std::ifstream ifs("exam2x.hddm");
   hddm_x::HDDM xrec;
   hddm_x::istream istr(ifs);
   istr >> xrec;
   hddm_x::CourseList course = xrec.getCourses();
   int total_courses =course.size();
   int total_enrolled = 0;
   int total_credits = 0;
   int total_passed = 0;
   hddm_x::CourseList::iterator iter;
   for (iter = course.begin(); iter != course.end(); ++iter) {
      if (iter->getResult().getPass()) {
         if (iter->getYear() > 1992) {
            total_credits += iter->getCredits();
         }
         ++total_passed;
      }
   }
   std::cout << course().getName() << " enrolled in "
             << total_courses << " courses "
             << "and passed " << total_passed << " of them, " << std::endl
             << "earning a total of " << total_credits
             << " credits." << std::endl;
   return 0;
}
```

Running the above code should produce output like the following:

```
Humphrey Gaston enrolled in 3 courses and passed 3 of them,
earning a total of 8 credits.
```

C. C++ API reference

The C++ code that defines the HDDM library for a particular data model is automatically generated by the tool `hddm-cpp`. The specific names for classes and members in the generated library code are taken from the user's data model. The following API description adopts a simple notational convention to document the actual classes and methods that are intended for use in application code. The HDDM class abbreviation for a particular data model is denoted by the wildcard '*', as in the C++ namespace `hddm_*`. A named element from the user's data model is denoted by *element* when written in lower case, and by *Element* when it is written in upper case, eg. in the context of its datatype (C++ class). The plural form *Elements* is used to refer to a list of *Element* objects. The name of an attribute is denoted *attribute* when it is used in lower case, and *Attribute* when upper case is required. The datatype of an attribute is denoted *Atype*, which represents one of the narrow set of simple data types that are supported by HDDM. For example, the expression "*Atype* hddm_*::*Element*List(n).get*Attribute*()" would translate in the context of a specific data model into the C++ expression "`int hddm_s::PhysicsEventList(n).getEventNo()`", where *Atype* has become `int`, *Element* has become the capitalized element name `PhysicsEvent` and *Attribute* has become its capitalized attribute name `EventNo`.

- `class hddm_*::ostream`
  Implements the output functions of native hddm streams.
  - `ostream(std::ostream &src)`
    Constructor of a hddm_*::ostream object which wraps a std::ostream streambuf and inserts itself

between the source of output data and the underlying stream that receives the streaming data. The destination may be a file (std::ofstream) which may be used in the call to this constructor, but it is not the only possibility. Standard output (std::cout) would be an equally valid argument to pass to this constructor.

- `ostream &operator<<(HDDM &record)`
  All output of data to HDDM output streams is via this method, using standard C++ semantics for streaming output: my_ostream ¡¡ my_record.
- `void setCompression(int flags)`
  Set the compression filter that is active on this output HDDM stream. The compression mode of a stream can be changed at any time in the flow, although normally it is set when the stream is first opened and left unchanged after that. The following values are supported for flags.
  - `k_no_compression`
  - `k_z_compression`
  - `k_bz2_compression`
- `int getCompression() const`
  Get the compression mode that is presently active on this output HDDM stream. The compression mode of a stream can be changed at any time in the flow. For the possible return values, see the constants listed above under setCompression().
- `void setIntegrityChecks(int flags)`
  Set the integrity checking that is presently active on this output HDDM stream. The integrity checking mode of a stream can be changed at any time in the flow, although normally it is set when the stream is first opened and left unchanged after that. Enabling integrity checking writes redundant information to the stream so that it can be checked for consistency when the data are read back later. The following values are supported for flags.
  - `k_no_integrity`
  - `k_crc32_integrity`
- `int getIntegrityChecks() const`
  Get the integrity checking that is presently active on this output HDDM stream. The integrity checking mode of a stream can be changed at any time in the flow, although normally it is set when the stream is first opened and left unchanged after that. Enabling integrity checking writes redundant information to the stream so that it can be checked for consistency when the data are read back later. Valid integrity checking modes are listed above under setIntegrityChecks().
- `streamposition getPosition()`
  Returns the current position of the underlying output stream at the time the method is called. This value can be saved and then passed back in later to hddm_*::istream::setPosition() to return to the same place and read from the stream from this position forward. Note that this functionality works even in the context of compressed streams.
- `int getBytesWritten() const`
  Returns the total number of bytes written by the user to the output HDDM stream since it was opened to the present time. Note that this may be different from the number of bytes written to the underlying output medium, if compression is active on the stream.
- `int getRecordsWritten() const`
  Returns the total number of HDDM records written by the user to the output HDDM stream since it was opened to the present time.

- `class hddm_*::istream`
  Implements the input functions of native HDDM streams.
  - `istream(std::istream &src)`
    Constructor of a hddm_*::istream object which wraps a std::istream streambuf and inserts itself between the user application and the underlying source stream that supplies the streaming data. The source may be a file (std::ifstream) which may be used in the call to this constructor, but that is not the only possibility. Standard input (std::cin) would be an equally valid argument to pass to this constructor.

- `istream &operator>>(HDDM &record)`
  All input of data from HDDM output streams is via this method, using standard C++ semantics for streaming input: my_ostream ¿¿ my_record;
- `void skip(int count)`
  Tells the input stream to skip forward by count records, and position itself at the start of the first record following.
- `int getCompression() const`
  Get the compression mode that is presently active on this input HDDM stream. The compression mode of a stream can change at any time in the flow. This information is encoded into the input stream, and can be sensed but cannot be changed at read time. For the possible return values, see the constants listed above under hddm_*::ostream::setCompression().
- `int getIntegrityChecks() const`
  Get the integrity checking mode that is presently active on this input HDDM stream. The integrity checking mode of a stream can change at any time in the flow. This information is encoded into the input stream, and can be sensed but cannot be changed at read time. Any time integrity checking information is present on an input stream, the integrity checks are automatically carried out, and exceptions are thrown any time the checks fail on input. For the possible return values, see the constants listed above under setCompression().
- `streamposition getPosition()`
  Returns the current position of the underlying output stream at the time the record was read from the stream. This is a bit different in behavior from getPosition on an output stream, which reports the current position, not the position prior to the last write operation. But it was decided that returning the position at the beginning of the current (most recently read) event is a more natural and useful behavior for this method than reporting the position following the most recent read. This value can be saved and then passed back in later to setPosition() to return to the same place and read from the stream from this position forward. Note that this functionality works even in the context of compressed streams.
- `void setPosition(const streamposition &pos)`
  Repositions the input stream to the position pos. The value of pos must be one that was returned from a previous call to getPosition on the same stream, either when it was being written or during a previous read. Supplying a trial value for pos in an attempt to find a random record in a file by guess-and-check will result in unpredictable behavior. Note that this functionality works even in the context of compressed streams.
- `int getBytesRead() const`
  Returns the total number of bytes read by the user from the input HDDM stream since it was opened to the present time. Note that this may be different from the number of bytes read from the underlying output medium, if compression is active on the stream, or if one or more calls have been made to setPosition.
- `int getRecordsRead() const`
  Returns the total number of HDDM records read by the user from the input HDDM stream since it was opened to the present time.
- `bool eof()`
  Returns true if the input stream is positioned at the end of the stream, otherwise false.
- `bool operator!()`
  Returns true if the state of the input stream such that another read may return valid data, otherwise false. The false condition may indicate either that the stream has reached the end, or an unrecoverable error has occurred and the stream is no longer readable.
- `operator void*()`
  Effectively acts as the contrary of operator!().

- `class` *Element*
  This interface is implemented in the generated HDDM C++ header file and implementation for each *Element* described by a tag in the user's data model xml document.
  - [no constructors]
    Element objects are not constructed directly by the user. They are produced by calling the factory

methods `add`*Elements*() on the container *element* under which they exist in the data model. That way, all *element*s are owned within the model hierarchy of one record, and are cleanly disposed of by the destructor of the record when it is deleted.

- `void clear()`
  Zeros the values of all attributes in the *element* and recursively deletes all *element*s contained as descendents within the data model.
- *Atype* `get`*Attribute*`() const`
  Returns the current value of *Attribute* with type *Atype* belonging to this *element*. This is the standard data getter method in HDDM.
- `const void *getAttribute(const std::string &name, hddm_type *atype=0) const`
  Returns the value of the attribute name together with its type if argument atype is present. If an attribute with this name does not exist in the data model for this *element* or any of its ancestors in the data model then a value 0 is returned. This method is provided to cover the use case when the name of the attribute is undetermined at compile time, so the get*Attribute*() method is not applicable. Note that there is no corresponding setAttribute(std::string name) method; to set the value of an attribute, the user needs a direct reference to the container *element*, and the name and type must be known at compile type. Defined values for the return argument atype are as follows.
  1. `int` - 32-bit signed integers
  2. `long` - 64-bit signed integers
  3. `float` - IEEE 754 binary32 floating-point numbers
  4. `double` - IEEE 754 binary64 floating-point numbers
  5. `boolean` - 0 (false) or 1 (true)
  6. `string` - ordered sequence of UTF-8 bytes
  7. `anyURI` - string conforming to the URI schema (see RFC 3305)
  8. `Particle_t` - string literal found in the dictionary of particle names (see hddm header file particles.h)
- `void set`*Attribute*`(`*Atype* `value)`
  Sets the current value of *Attribute* with value of type *Atype*. This is the standard data setter method in HDDM.
- *Element* `&get`*Element*`([int index=0])`
  Returns a reference to a contained *Element*. If exactly one *element* of this type is specified as belonging to the containing *element* in the data model then the index argument is not supported, and the call returns a reference to the one contained object. If a variable number of *Element*s is specified as belonging to the containing *element* in the data model then the index argument must be less than the total number of such *Element*s actually contained. A call to this method with an invalid index produces unpredictable results, and is not supported. In any case where the number of *Element*s may not be known, the next method is safe and is actually recommended in most cases.
- *Element*`List &get`*Elements*`()`
  Returns an iterable *Element*List containing an ordered list of *Element* objects contained within this *element*. This method is valid even for cases where the container *element* is specified to have exactly one contained *Element*. The *Element*List is a lightweight container of *Element* objects that supports the full semantics of std::list¡*Element*¿ with some extensions. See below for more details on the *Element*List class and its associated iterators.
- *Element*`List add`*Elements*`(int count=1, int start=-1)`
  Adds count newly initialized *Element*s to this container *element*. This is the standard factory method for adding data to the HDDM record. If there are already one or more *Element*s associated to this container *element*, the start argument specifies where to insert the new ones in the existing list, so that they appear immediately before the existing *element* at position start. If start is -1 (or not given), the new *Element*s are placed at the end of the list. Calling add*Elements* with an invalid value of start gives unpredictable results. The return value is an *Element*List containing the new *element*s just added, not the full list just extended. To get the full list, a subsequent call to get*Elements*() is needed (see above).

- `void delete`*Elements*`(int count=-1, int start=0)`
  Delete count *Element*s from this container *element*, starting at position start. Calling delete*Elements*() with count = -1 results in all *element*s from position start onward being deleted. Calling delete*Elements*() without arguments deletes all *Element*s from the container *element*.
- `std::string toString(int indent=0)`
  Returns a printable string representation of this *Element*, including recursively all of its content.
- `std::string toXML(int indent=0)`
  Returns a plain-text xml representation of this *Element*, including recursively all of its content.
- `class` *Element*`List`
  An *Element*List class is defined in the HDDM header file for each user *Element* defined in the data model. It is a lightweight derivation of std::list that can be used to efficiently iterate over the contents of the HDDM record. *Element*List is a public descendant of std::list so any of the standard semantics for std::list iterators also apply to these. In particular, the following std::list methods are overloaded to provide specific functionality related to lists of HDDM elements. *Element*Lists are produced by members of the data model element hierarchy owned by a HDDM record, so mutating methods acting on the *Element*List (see below) automatically act on the container *element* they are derived from.
- `bool empty() const`
  Returns true if the list is empty, otherwise false [standard].
- `int size() const`
  Returns the number of members contained in the list [standard].
- *Element* `&front() const`
  Returns a reference to the first member of the list [standard]. Calling front() on an empty list produces unpredictable results.
- *Element* `&back() const`
  Returns a reference to the last member of the list [standard]. Calling last() on an empty list produces unpredictable results.
- *Element* `&operator()()`
  Returns a reference to the first member of the list [extension]. Calling operator() on an empty list produces unpredictable results.
- *Element* `&operator()(int index)`
  Returns a reference to member index of the list [extension]. Calling operator(i) on a list with i or fewer members produces unpredictable results.
- *Element*`List::iterator begin() const`
  Returns an iterator pointing to the start of the list [standard].
- *Element*`List::iterator end() const`
  Returns an iterator pointing to the end of the list [standard].
- `void clear()`
  Recursively deletes all members of the list [extension]. Note that this affects not only the immediate *Element*List object, but also the container *element* from which this list was extracted. Normally users would not call clear() on an *Element*List that was returned from one of the factory methods of a container *element*. Instead, calling delete*Elements*() on the container *element* would achieve the same result, and lead to more readable code.
- *Element*`List add(int count=1, int start=-1)`
  Adds count newly initialized members of *Element* to the list [extension]. If the list already contained one or more *Element*s, the start argument gives the position start before which the additional *Element*s should appear in the updated list. A value start = -1 (or omitted) results in the new *Element*s being added to the end of the list. Calling add with a negative count or invalid start has unpredictable results. Note that this affects not only the immediate *Element*List object, but also the parent HDDM record from which this list is derived. Normally users would not call add() on an *Element*List that was previously returned from one of the factory methods of a container *element*. Instead,calling add*Elements*() on the containing *element* would achieve the same result, and lead to more readable code.

  - `void del(int count=-1, int start=0)`
    Deletes count members of *Element* from the list [extension]. *Element*s are deleted starting at position start. A value count = -1 (or omitted) results in all of the *Element*s at start and following being deleted. Note that this affects not only the immediate *Element*List object, but also the containing *element* from which this list is derived. Normally users would not call del on an *Element*List that was previously returned from one of the factory methods of a container *element*. Instead,calling delete*Elements*() on the containing *element* would achieve the same result, and lead to more readable code.
  - *Element*`List slice(int first=0, int last=-1)`
    Returns a new *Element*List object containing a contiguous subset of the original list on which this method is called. The limits of the subset are specified by first and last arguments. A call with first == last results in a list of length 1. Calling slice with first or last outside the bounds of the list gives unpredictable results.
  - `std::string toString(int indent=0)`
    Recursively calls toString() on the *element*s of the list, and returns the results concatenated into a single string.
  - `std::string toXML(int indent=0)`
    Recursively calls toXML() on the *element*s of the list, and returns the results concatenated into a single string.

- `class` *Element*`List::iterator` or *Element*`List::const_iterator`

  - *Element* `*operator->() const`
    Returns a pointer to the list iterator item *Element* [standard].
  - `T &operator*() const`
    Returns a reference to the list iterator item *Element* [standard].
  - `iterator operator+=(int offset)`
    Increments the list iterator by offset [extension].
  - `iterator operator-=(int offset)`
    Decrements the list iterator by offset [extension].
  - `iterator operator+(int offset) const`
    Adds offset to the list iterator and returns a new iterator [extension].
  - `iterator operator-(int offset) const`
    Subtracts offset to the list iterator and returns a new iterator [extension].
  - `int operator-(iterator iter) const`
    Takes the difference of two list iterators of the same type and returns the count. The two iterators must be initialized from the same *Element*List, or the results are unpredictable.
  - `class hddm_*::HDDM`
    This is the top-level *Element* from the data model, so as such it inherits all of the methods described above for class hddm_*::*Element*. In addition, it supports the following special methods that are only provided by the top-level record element.
  - *Element*`List &get`*Elements*`()`
    Returns an iterable *Element*List containing an ordered list of all *Element* objects contained within this record, regardless of whether or not they are immediate contents of the top-level element. This is a special case of the more general method of the same name that applies only to those *Element* classes that have *Element*s as their direct descendants in the data model. In this way, one can get a complete list of any particular type of *Element* that appears anywhere in the data model hierarchy for a particular record, without having to traverse the entire tree to find them. Such a get*Elements* method is provided by the top-level HDDM record for all instances of *Element* in the data model.
  - `static std::string DocumentString()`
    Returns the data model metadata for this HDDM class library as a complete xml document contained in a single printable string. The document string contains embedded newlines so that it looks readable when printed or written to a text file. This plain text string is the first thing in every HDDM file, which makes it easy to inspect their contents by simply printing everything up to the final ¡/HDDM¿ tag that marks the end of the HDDM document string.

VI. HDDM IN C

If you have access to a HDDM file that was already written, copy it into your work directory and use it as a template for building a python module to access the model data as c struct records. Otherwise, the HDDM package distribution directory contains a simple example in `models/exam1x.hddm` that you can use for this purpose. The following commands build the c library that you will need to read and write HDDM streams that conform to this template.

```
$ hddm-c exam1x.xml
$ gcc -c hddm_x.c -I $HDDM_INSTALL_DIR/include \
  -L $HDDM_INSTALL_DIR/lib64 -lxstream -lz -lbz2
$ ar -r libhddm_x.a hddm_x.o
```

A. writing HDDM files in c

This example turns once again to the template `exam1x.hddm` that is included with the source distribution. Use the instructions in the previous section to build the `hddm_x` c API library, then create a new main program source file and cut/paste the code below into it, then save it.

```
#include "hddm_x.h"

int main()
{
   x_iostream_t* fp;
   x_HDDM_t* exam2;
   x_Student_t*  student;
   x_Enrolleds_t* enrolleds;
   x_Courses_t* courses;
   x_Result_t* result;
   string_t name;
   string_t grade;
   string_t course;

   // first build the complete nodal structure for this record
   exam2 = make_x_HDDM();
   exam2->student = student = make_x_Student();
   student->enrolleds = enrolleds = make_x_Enrolleds(99);
   enrolleds->mult = 1;
   enrolleds->in[0].courses = courses = make_x_Courses(99);
   courses->mult = 3;
   courses->in[0].result = make_x_Result();
   courses->in[1].result = make_x_Result();
   courses->in[2].result = make_x_Result();

   // now fill in the attribute data for this record
   name = malloc(30);
   strcpy(name,"Humphrey Gaston");
   student->name = name;
   enrolleds->in[0].year = 2005;
   enrolleds->in[0].semester = 2;
   courses->in[0].credits = 3;
   course = malloc(30);
   courses->in[0].title = strcpy(course,"Beginning Russian");
   grade = malloc(5);
   courses->in[0].result->grade = strcpy(grade,"A-");
   courses->in[0].result->Pass = 1;
   courses->in[1].credits = 1;
   course = malloc(30);
   courses->in[1].title = strcpy(course,"Bohemian Poetry");
   grade = malloc(5);
   courses->in[1].result->grade = strcpy(grade,"C");
   courses->in[1].result->Pass = 1;
   courses->in[2].credits = 4;
   course = malloc(30);
   courses->in[2].title = strcpy(course,"Developmental Psychology");
   grade = malloc(5);
   courses->in[2].result->grade = strcpy(grade,"B+");
   courses->in[2].result->Pass = 1;

   // now open a file and write this one record into it
   fp = init_x_HDDM("exam2.hddm");
   flush_x_HDDM(exam2,fp);
   close_x_HDDM(fp);

   return 0;
}
```

Save this c program to a file called `write_exam2.c` and compile it into an executable using a command like the following.

```
$ gcc -o write_exam2 write_exam2.c hddm_x.o -I. -I $HDDM_INSTALL_DIR/include \
  -L $HDDM_INSTALL_DIR/lib64 -l xstream -lbz2 -lz
```

The shell environment variables containing the package installation paths in the above compile command may need to be customized for your own environment. Once it completes successfully, you will find the executable `write_exam2` in the working directory. Run it as `./write_exam2` and it should create a new HDDM file called `exam2.hddm`. Running `hddm-xml write_exam2.hddm` should produce output like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<HDDM class="x" version="1.0" xmlns="http://www.gluex.org/hddm">
  <student name="Humphrey Gaston">
    <enrolled semester="2" year="2005">
      <course credits="3" title="Beginning Russian">
        <result Pass="1" grade="A-" />
      </course>
      <course credits="1" title="Bohemian Poetry">
        <result Pass="1" grade="C" />
      </course>
      <course credits="4" title="Developmental Psychology">
        <result Pass="1" grade="B+" />
      </course>
    </enrolled>
  </student>
</HDDM>
```

This example explains most of what you need to know to set up HDDM c-structs in memory, and write them to an output file. All storage for HDDM data is allocated on the heap. Most of this allocation is carried out automatically by the `make_x_XXXs()` functions, although for strings (char arrays) the user needs to allocate initial storage for the values. Memory pointed to by the pointers returned by the `make_x_XXXs()` functions is owned by the user code until the pointer to it gets assigned to a HDDM struct member that is designated in the data model to hold it. After that, the memory is owned by the top-level HDDM container record object, and should only be freed by calling the `flush_x_HDDM()` method. Calling `flush_x_HDDM(record, fp)` with its second argument (FILE*) open to an output file causes the record to be written to the output file. Calling it as `flush_x_HDDM(record,0)` causes it to bypass the output serialization step. Either way, `flush_x_HDDM()` frees all memory owned by the HDDM record, discarding its contents, before it returns.

The example begins by creating an empty record by calling `make_x_HDDM()`. Then it populates the structure top-down by calling `make_x_XXXs()` for each tag XXX and assigning pointers to each one into the appropriate structure element of the parent element, where XXXs is the name of the tag element in the template transformed into a capitalized-plural form. The `add`XXXs() methods take a single optional int argument, which is the number of copies of that element that need to be added (default is 1). They return a pointer to an array of struct pointers which can be indexed in the usual c-fashion to access the individual members of the array. Each of the contained elements within a given host tag have a corresponding pointer in the host struct that must be assigned in the user code to the value returned by the `make_x_XXXs()` function, as illustrated. Any such pointers that are not assigned remain null (initialized by `make_x_XXXs`) and represent parts of the template tree that are missing from the record. This is a perfectly valid HDDM record, but user code must check for the NULL pointer condition before trying to dereference it since c has no automatic checking of the validity of pointers. As soon as a new struct array element is created, you can fill in the values of its attribute members using direct assignment semantics, as illustrated in the example above. Any values that are not explicitly assigned remain at the default values, typically zero or null.

B. reading HDDM files in c

This section assumes that you have created the file exam2.hddm using the instructions in the previous section. The following c program opens this file and extracts bits of information from the first record,

writing a summary report at the end. Of course, in actual practice a HDDM file would probably contain many records, and the analysis would loop over many instances student.

```
#include "hddm_x.h"

int main()
{
   x_iostream_t* fp;
   x_HDDM_t* exam2;
   x_Student_t* student;
   x_Enrolleds_t* enrolleds;
   int enrolled;
   x_Courses_t* courses;
   int course;
   int total_enrolled,total_courses,total_credits,total_passed;

 // read a record from the file
   fp = open_x_HDDM("exam2.hddm");
   if (fp == NULL) {
      printf("Error - could not open input file exam2.hddm\n");
      exit(1);
   }
   exam2 = read_x_HDDM(fp);
   if (exam2 == NULL) {
      printf("End of file encountered in hddm file exam2.hddm, quitting!\n");
      exit(2);
   }

   // examine the data in this record and print a summary
   total_enrolled = 0;
   total_courses = 0;
   total_credits = 0;
   total_passed = 0;
   student = exam2->student;
   enrolleds = student->enrolleds;
   total_enrolled = enrolleds->mult;
   for (enrolled=0; enrolled<total_enrolled; ++enrolled) {
      courses = enrolleds->in[enrolled].courses;
      total_courses += courses->mult;
      for (course=0; course<courses->mult; course++) {
         if (courses->in[course].result->Pass) {
            if (enrolleds->in[enrolled].year > 1992) {
               total_credits += courses->in[course].credits;
            }
            ++total_passed;
         }
      }
   }
   printf("%s enrolled in %d courses.\n",student->name,total_courses);
   printf("He passed %d of them, earning a total of %d credits.\n",total_passed,total_credits);

   flush_x_HDDM(exam2,0);  // don't do this until you are done with exam2
   close_x_HDDM(fp);
   return 0;
}
```

Running the above code should produce output like the following:

```
Humphrey Gaston enrolled in 3 courses and passed 3 of them,
earning a total of 8 credits.
```

Having read the section above on how to write HDDM records using the c interface, it should be easy to understand the meaning of the above code. The `read_x_HDDM()` call allocates all of the memory needed to stand up the full record hierarchy in memory. The `flush_x_HDDM(record,0)` call at the end of the loop ensures that all of this memory gets recycled to the heap before the next record is read in. Accessing leaf elements that are deep inside the HDDM template hierarchy can only be achieved by traversing all of the nodes in the tree above, which makes a simple data mining operation somewhat verbose, as illustrated in the above example, although it still scales well because the model is hierarchical, not a linked list. If you are unsure about how to do something, browsing within the header file is probably not going to be very rewarding because all of the internal functionality of the logic that supports the serialization/deserialization of the data is exposed there. However, the user API is very simple. Access to the data-bearing attributes is through direct struct member access. Only the `make_x_XXXs` functions and the input/output functions (open, close, read, flush, skip) should be called by the user; all the rest are for internal use. As is the case for for all of the other API's, the template itself should be the only documentation you need to consult when writing code that interacts with HDDM data.

C. c API reference

The c API is no longer in active development. It is supported only for legacy applications that rely on it. The features described in the VII Advanced Features section below are not available using the c API. The only things that are ensured with regard to ongoing support of the c API is that it can read the streams that it writes based on any valid HDDM template, and that HDDM files written using the c-API can be read by applications built using any of the other API's. The converse of the last statement is not guaranteed to be true in all cases. If an input file is not readable by the c-API, it prints a polite error message reporting this fact and exits.

VII. ADVANCED FEATURES

A. on-the-fly compression/decompression

HDDM streams added support for on-the-fly compression on output (and decompression on input) with the introduction of the C++ API. Because the python API is a thin wrapper around the C++ classes, it also supports this feature. Compression can obviously only be controlled when the stream is being written. It can be switched on and off at any time after the stream is opened, either before the first record is written or any time thereafter. Whenever it is turned on or off, a small marker is written to the byte stream that tells the reader when to enable/disable decompression on the input stream. These transitions occur silently during input; no user action is needed, and no log messages are automatically generated. Two compression options are supported.

1. **bzip2 compression** - This option offers the best compression ratio, a factor of about 2.5 for particle physics experimental Monte Carlo data. It is also the most expensive in terms of cpu time needed during output. Cpu demand for decompression is much lower, more than an order of magnitude.

2. **zlib compression** - This option offers somewhat lower compression ratios, a factor of about 1.9 for particle physics experimental Monte Carlo data. However, it is also much less expensive in terms of cpu time than bzip2, by more than a factor 3. Cpu demand for decompression is much lower than compression, as is usually the case with codecs.

Both options are provided because each has its strengths and weaknesses in terms of cost/performance, and their relative behaviors may be quite different for different data models. Another factor to take into consideration when deciding which compression algorithm to use, if at all, is the implications of the compression block size on the efficiency for random access to records in the stream. For more information about random access, see the relevant section below. If the stream is uncompressed, random access to a particular record generates a read starting at the beginning of that specific record and only taking in the contents of that record, whereas if the stream is compressed, the entire compression block containing the record

must be decompressed before the data for the desired record can be pulled in. The compression blocks for bzip2 compression are almost 1MB in size, whereas the zlib blocks are much smaller, around 32KB. There is no general answer to the question of which compression option is best. The person producing the data should consider what the most likely scenarios are for reading the data, and weigh the costs and benefits of compression before making this decision.

In the C++ API, the HDDM namespaces have defined the following constants to distinguish different states of compression:

- `k_no_compression`
- `k_z_compression`
- `k_bz2_compression`

One of these three constants should be passed as mode to the `setCompression(mode)` method of the `hddm_x::ostream` class to initialize or change the compression state of any given output stream. All records written after this method is called will reflect the change. The present compression mode of either an input or output hddm stream can be queried by calling method `getCompression()`. The return value (int) can be compared with the three constants above to determine which of the three modes is presently enabled.

In python HDDM modules, stream objects of class `hddm_x.istream` and `hddm_x.ostream` support selection and sensing of the current compression mode by exposing read/write attribute `compression`. The named constants listed above are defined within the `hddm_x` module namespace. Setting bz2 compression on an open ostream would look like, `fout.compression = hddm_x.k_bz2_compression`.

B. on-the-fly data integrity checks

HDDM streams added support for on-the-fly data integrity checks with the introduction of the C++ API. Because the python API is a thin wrapper around the C++ classes, it also supports this feature. Data integrity verification works by the writer computing a hash value on each output record and storing it as part of the output stream, which the reader then pulls off the stream and uses to verify the integrity of the data is reads from the stream. Two 32-bit hash algorithms are currently supported by HDDM.

1. **CRC32** - the 32-bit cyclic redundancy check algorithm
2. **MD5** - the MD5 one-way hash algorithm

CRC is considered in cryptographic circles as an error detection algorithm, meaning that a single bit change in the data record will result in a change in the 32-bit code, and it is very rare that a combination of errors cancels out and generates the same crc as the original data. This is probably all we need for our data, and it is much faster to compute than MD5. MD5 is called a one-way hash in cryptographic jargon, which means that a single bit change in the data record will be reflected in a *vastly* different value for this 32-bit code, with approximately 50% of the bits changing in the hash as a result of a single bit-flip in the input. One might consider this marginally better for error detection in a random byte stream, but it is more expensive to compute than a CRC code. Neither MD5 nor CRC32 options result in any noticeable overhead in the context of any models tested so far.

In the C++ API, the HDDM namespaces have defined the following constants to distinguish different states of data integrity checking:

- `k_no_integrity`
- `k_crc32_integrity`
- `k_md5_integrity`

One of these three values should be passed as mode to the `setIntegrityChecks(mode)` method of the `hddm_x::ostream` class to change the current state of the output stream. All records written after this method is called will reflect this change. The present integrity checking mode of either an input or output HDDM stream can be queried by calling method `getIntegrityChecks()`. The return value (int) can be compared with the three constants above to determine which of the three integrity checking modes is presently enabled.

In python HDDM modules, stream objects of class `hddm_x.istream` and `hddm_x.ostream` support the same interface by exposing read/write attribute `integrity`. The named constants listed above are defined within the `hddm_x` module namespace. Setting CRC32 compression on an open ostream might look like, `fout.integrity = hddm_x.k_crc32_integrity`.

C. random access to HDDM records

HDDM streams added support for random access on input with the introduction of the python API. Because the python API is a thin wrapper around the C++ classes, it is also supported by the C++ API. Random-access writing to HDDM streams is not supported; the access point for output streams is always positioned after the end of the previous output record. Random-access reads are supported on any input stream that supports repositioning. To succeed, the random access position must have been generated by a previous call to the `getPosition()` method of the same HDDM stream, either during the initial phase when the stream was being written, or during a subsequent read pass over the same stream. The `getPosition()` query returns an opaque value representing a point in the stream either at the beginning of the first record, or immediately after the last valid record read or written on the stream. Random access to individual records in the input HDDM stream can take place in any order, and involve displacements either forward or reverse from the position of last access.

Attempts to access a stream at an uninitialized position, or at a position that was generated on a different HDDM stream, will result in unpredictable behavior, most likely a segmentation fault upon the next attempt to read from the stream. The following three integer values are needed to define a stream position.

1. **block_start** (uint64_t) - absolute stream position (std::streampos value) of the beginning of the block containing the record
2. **block_offset** (uint32_t) - offset with the block to the start of the designated record, or 0 if compression is disabled
3. **block_status** (uint32_t) - complete state (compression, integrity, other information about the stream state at this position)

If a database were used to store a map of valid positions for a set of HDDM files, a minimum field width of 128 bits would be needed. Of course, you might want to save the name and creation date of the input file that the positions apply to, so that you do not accidentally try to apply them to a different file than they were created for. If the stream is uncompressed then `block_offset`=0, but still `block_start` and `block_status` would be needed. The `block_status` value is typically the same for all positions in a given file or dataset, so in most cases only a single value for that variable needs to be kept, together with a list of the starts and offsets for the given file.

The object class `hddm_x::streamposition` is used to hold stream position information. Public data members with the names listed above are exposed for members of the streamposition class. Both `hddm_x::istream` and `hddm_x::ostream` classes have `getPosition()` members that return a streamposition value. It can either be recorded by saving the values of its three data members, or by keeping the object in memory and passing it to the corresponding `istream::setPosition(streamposition)` method called on an istream that is (presumably) open for input on the same file. If the 3 values are stored, they can later be quickly turned back into a streamposition object using the constructor `streamposition(start,offset,status)`.

HDDM files that were written since this feature was introduced are marked with the capability to support random access. To check if a given file that has been opened for input on a `hddm_x::istream` supports random access, simply call method `getPosition()` within a try-catch block and catch the RuntimeError that is thrown if the input does not support this feature.

Support in the python API for random access follows closely the scheme described above for C++. The `hddm_x.istream` and `hddm_x.ostream` classes both have read/write data members called `position` that reference objects of type `hddm_x.streamposition`. These objects can be saved and then later assigned to an `hddm_x.istream` opened on the same file to seek to the same position in the input stream using a command like `fin.position = hddm_x.streamposition(start,offset,status)`. Until another position assignment is executed, reading proceeds in a serial fashion starting from the last record read from the stream.

VIII. MULTITHREADED I/O WITH HDDM

The HDDM i/o library is thread-safe, meaning that it is permitted to read or write simultaneously from multiple threads within a single application without interlocks. Contention for access to the underlying data source or sink is taken care of automatically by the library. In fact, users are encouraged to use multithreading to do overlapping i/o operations in order to take advantage of parallelism in the compression/decompression code.

IX. SUPPORT FOR HDF5 FILE FORMAT

When selecting a binary format for HDDM, there were several open standards that were considered. One that was very close conceptually to HDDM was BinX [1] but the implementation of BinX was not complete at the time HDDM was being developed, and the future of support for BinX or the extent of community adoption was unclear at the time. Also the generality of BinX in its capability of representing an arbitrary well-formed XML document limited the potential optimizations that could be used for the regularly repeating structures that are present in data from modern scientific instruments.

Another option that was consider during the design of HDDM was to use the HDF file format. HDF was already quite mature at the time, with broad community acceptance, and it was well-optimized for encoding the repeated structures of scientific data. However, HDF had the limiting constraint that all structures have fixed length at all but the highest level of the hierarchy. This one limitation made it very difficult to encode structures with variable-length lists at all levels of the hierarchy, which led to the decision for HDDM to have its own custom binary encoding based on XDR [2].

Since the time that decision was made, the HDF standard has evolved into HDF5 [3], and support for ragged arrays has been introduced. This has made it possible to add support in HDDM for HDF5 file encoding without any changes to the HDDM data access API. Although it is not as efficient or compact as the HDDM native format, HDF5 encoding offers important advantages as a public format for publishing, archival, interchange of scientific data; widely used on science portals. HDF5 has persistent community support, is well documented, and has extensive tunability for optimized i/o on a wide variety of platforms from desktops to HPC facilities.

HDF5 is not a replacement for HDDM because it is only an i/o library. By contrast, HDDM contains an i/o library, but also supports data inspection / mutation C++ semantics, similar to a STL container. Adding the capability to layer HDF5 under HDDM provides the installed HDDM application codebase the ability to read and write data in a standard archival format with very little additional work for users. HDF5 supports indexed files (random-access i/o) which means one can get rapid access to random events without reading the entire stream.

Adding HDF5 i/o functionality to existing HDDM applications is relatively simple: replace the existing HDDM i/o semantics in a C++ application reading from a HDDM file,

```
std::ifstream hddm_file("my_sims.hddm");
hddm_s::istream hddm_in(hddm_file);
hddm_s::HDDM record;
while (hddm_in >> record) { do_your_thing(record); }
hddm_in.close();
```

with

```
hid_t hdf5_in = hddm_s::HDDM::hdf5FileOpen("my_sims.hdf5");
hddm_s::HDDM record;
while (record.hdf5FileRead(hdf5_in) == 0) { do_your_thing(record); }
hddm_s::HDDM::hdf5FileClose(hdf5_in);
```

Similarly, for a C++ application writing to a HDDM file, replace

```
std::ofstream hddm_file("my_sims.hddm");
hddm_s::ostream hddm_out(hddm_file);
hddm_s::HDDM record;
while ( do_your_thing(record) ) { hddm_out << record; }
hddm_out.close();
```

with

```
hid_t hdf5_out = hddm_s::HDDM::hdf5FileCreate("my_sims.hdf5");
hddm_s::HDDM record;
while ( do_your_thing(record) ) { record.hdf5FileWrite(hdf5_out); }
hddm_s::HDDM::hdf5FileClose(hdf5_out);
```

A more complete description of the HDF5 additions to the HDDM C++ API is given in the following section.

A similar set of extensions have been introduced to the HDDM python module to support HDF5 i/o semantics. For a python application that reads from a HDDM file, simply replace

```
import hddm_s
for record in hddm_s.istream("my_sims.hddm"):
   do_your_thing(record)
```

with

```
import hddm_s
record = hddm_s.HDDM()
hdf5_in = hddm_s.hdf5FileOpen("my_sims.hdf5")
while record.hdf5FileRead(hdf5_in) == 0:
   do_your_thing(record)
hddm_s.hdf5FileClose(hdf5_in)
```

Similarly, for a python application writing to a hddm file, replace

```
import hddm_s
record = hddm_s.HDDM()
hddm_out = hddm_s.ostream("my_sims.hddm")
while do_your_thing(record):
   hddm_out.write(record)
hddm_out.close()
```

with

```
import hddm_s
record = hddm_s.HDDM()
hddm_out = hddm_s.hdf5FileCreate("my_sims.hddm")
while do_your_thing(record):
   record.hdf5FileWrite(hddm_out)
hddm_out.close()
```

A. C++ API elements for HDF5

- `static hid_t hddm_*::HDDM::hdf5FileCreate(std::string name, unsigned int flags=0)`
  Opens a new HDF5 file name for writing with access flags, and returns the HDF5 identifier for the new file. The following values are supported for flags, defined in H5Fpublic.h:
  - `H5F_ACC_TRUNC` (default) - truncate file, if it already exists, erasing all data previously stored in the file
  - `H5F_ACC_EXCL` - fail if file already exists.

  An implicit call to hdf5FileStamp is made by hdf5FileCreate so that any records subsequently written with hdf5FileWrite will be stamped with the appropriate HDDM metadata for this class. Failure is indicated by a negative value in the returned identifier.

- `static hid_t hddm_*::HDDM::hdf5FileOpen(std::string name, unsigned int flags=0)`
  Opens an existing HDF5 file name for reading with access flags, and returns the HDF5 identifier for the file. The following values are supported for flags, defined in H5Fpublic.h:
  - `H5F_ACC_RDONLY` (default) - allow read-only access to file.
  - `H5F_ACC_RDWR` (default) - allow read and write access to file.

An implicit call to hdf5FileCheck is made by hdf5FileOpen so that any records subsequently read with hdf5FileRead are validated against the appropriate HDDM metadata for this class. Failure is indicated by a negative value in the returned identifier.

- `static herr_t hddm_*::HDDM::hdf5FileClose(hid_t file_id)`
  Closes an HDF5 file that has been previously opened by hdf5FileOpen or hdf5FileCreate. Failure is indicated by a negative returned value.

- `herr_t hddm_*::HDDM::hdf5FileWrite(hid_t file_id, long int entry=-1)`
  Writes this record to the HDDM dataset at the output HDF5 location identified by file_id. The file_id may be either the value returned from a call to HDDM::hdf5FileCreate, or the HDF5 identifier of any HDF5 file or group within a file that was opened for writing. The optional argument entry is the record number offset from the start of the output dataset where this record should be written. A value of -1 for entry (or omitted) causes the write to take place at the current offset in the file, which is one past the last record written or the start of the dataset, if this is the first write. For the output record to be readable later through this API, the output file or group must be stamped for hddm output, but this is not checked by hdf5FileWrite. Output hddm record stamping is performed automatically by HDDM::hdf5FileCreate, but may also be done manually using HDDM::hdf5FileStamp if the user independently manages the opening and closing of HDF5 groups and files.

- `herr_t hddm_*::HDDM::hdf5FileRead(hid_t file_id, long int entry=-1)`
  Reads a record from the HDDM dataset at the input HDF5 location identified by file_id. The file_id may be either the value returned from a call to HDDM::hdf5FileOpen, or the HDF5 identifier of any HDF5 file or group within a file that was opened for reading. The optional argument entry is the record number offset from the start of the output dataset where this record should be read. A value of -1 for entry (or omitted) causes the read to take place at the current offset in the file, which is one past the last record read or the start of the dataset, if this is the first read. For the input record to be readable, this file or group must have been stamped for hddm input (see HDDM::hdf5FileCheck), but this is not checked by hdf5FileRead.

- `static herr_t hddm_*::HDDM::hdf5FileStamp(hid_t file_id, char **tags=0)`
  Creates or updates the HDDM metadata stamp for this HDDM class in the HDF5 output file location given by file_id. The output file is presumed to already have been opened for writing. The tags argument is an optional zero-terminated list of user-defined strings that are appended to the HDDM metadata stamp before it is written. These can be used later when the dataset is read (see hdf5FileCheck) to verify that the dataset meets certain user-defined requirements, in addition to the basic HDDM metadata stamp validation.

- `static herr_t hddm_*::HDDM::hdf5FileCheck(hid_t file_id, char **tags=0)`
  Checks the HDDM metadata stamp for this HDDM class in the HDF5 input file location given by file_id. The input file or group is presumed to already have been opened for writing. The tags argument is an optional zero-terminated list of user-defined strings that are checked against the HDDM metadata stamp that was saved when the dataset was written. These can be used later to verify that the dataset meets certain user-defined requirements, in addition to the basic HDDM metadata stamp validation.

- `static std::string hddm_*::HDDM::hdf5DocumentString(hid_t file_id)`
  Returns the HDDM metadata string that is saved in the currently open HDF5 location file_id. Normally the user does not need to be concerned with the literal meaning of this string, only that its value matches what is returned by HDDM::DocumentString() for this HDDM class.

- `static long int hddm_*::HDDM::hdf5GetEntries(hid_t file_id)`
  Returns the current number of entries in the HDDM dataset presumed to already exist in HDF5 location file_id, already presumed to be open. The file_id may be currently open for either reading or writing. If it is open for writing then at least one record must already have been written, otherwise this command will fail because the dataset does not yet exist in the file.

- `static herr_t hddm_*::HDDM::hdf5SetChunksize(hid_t file_id, hsize_t chunksize)`
  Sets the chunk size for output to the HDF5 location given by file_id, presumed to be already open for writing. For information about the concept of chunks in HDF5 files, see the HDF5 documentation. The HDF5 standard requires that the chunk size must be set before the first output record is written to the file, otherwise the request to change the chunk size is ignored. The default chunk size for

HDDM output is set when the file is opened to the value of HDF5_DEFAULT_CHUNK_SIZE in the hddm_*.hpp header file. At the time of this writing, the default value is 100.

- `static hsize_t hddm_*::HDDM::hdf5GetChunksize(hid_t file_id)`
  Returns the chunk size that was used for writing the HDDM dataset to the HDF5 location given by file_id. The file_id may be currently open for either reading or writing. If it is open for writing then at least one record must already have been written, otherwise this command will return the default chunk size because the dataset does not yet exist in the file. For information about the concept of chunks in HDF5 files, see the HDF5 documentation.

- `static herr_t hddm_::HDDM::hdf5SetFilters(hid_t file_id, std::vector<H5Z_filter_t> &filters)`
  Assigns an ordered list of HDF5 filters for use in subsequent writes to the HDF5 location file_id. Filters are used by the HDF5 library mainly for data compression and integrity checks. Each is identified by a unique integer that is registered for that filter with the HDF5 library. Filters are configurable by the user only when the dataset is written. When the dataset is read back later, the library automatically selects the correct set of filters so that valid data can be read back into memory. Any valid filter identifier supported by the local build of the HDF5 library can be included in the filters list. The following known filter identifier constants are defined within the HDDM header for convenience. By default, no filters are configured for HDDM output datasets.

  - `k_hdf5_gzip_filter` - gzip standard compression provided by hdf5
  - `k_hdf5_szip_filter` - szip standard compression provided by hdf5
  - `k_hdf5_bzip2_plugin` - bzip2 lossless compression used by PyTables
  - `k_hdf5_blosc_plugin` - Blosc lossless compression used by PyTables
  - `k_hdf5_bshuf_plugin` - bitshuffle shuffle filter at bit level instead of byte level
  - `k_hdf5_jpeg_plugin` - JPEG-XR compression filter used in jpeg images
  - `k_hdf5_lz4_plugin` - LZ4 fast lossless compression algorithm
  - `k_hdf5_lzf_plugin` - LZF fast lossless compression used by H5Py project
  - `k_hdf5_lzma_plugin` - modified LZMA compression filter (MAFISC)
  - `k_hdf5_zfp_plugin` - zfp rate, accuracy, or precision bounded compression for arrays of floats

- `static herr_t hddm_::HDDM::hdf5GetFilters(hid_t file_id, std::vector<H5Z_filter_t> &filters)`
  Gets the ordered list of HDF5 filters that are configured for i/o of HDDM datasets to the HDF5 location file_id. Filters are used by the HDF5 library mainly for data compression and integrity checks. Each is identified by a unique integer that is registered for that filter with the HDF5 library. For a partial list of supported filters, see the API description under hdf5SetFilters.

B. python API elements for HDF5

- `hddm_*.hdf5FileCreate(name, flags=0)`
  Create a new HDF5 file name, and open for writing HDDM records. For the meaning of the unsigned integer argument flags, see the corresponding method description under the C++ HDDM user API. Return value is the HDF5 location identifier for use in subsequent HDF5 i/o and dataset query operations.

- `hddm_*.hdf5FileOpen(name, flags=0)`
  Open an existing HDF5 file for reading HDDM records. For the meaning of the unsigned integer argument flags, see the corresponding method description under the C++ HDDM user API. Return value is the HDF5 location identifier for use in subsequent HDF5 i/o and dataset query operations.

- `hddm_*.hdf5FileClose(file_id)`
  Close an open HDF5 file and free its HDF5 resources. Return value is zero if the operation succeeded, negative if it failed.

- `record.hdf5FileWrite(file_id, entry=-1)`
  Perform a random-access write from this HDDM record to an output HDF5 file identified by file_id. The record is written to the output dataset at offset entry, or one past the last written record if entry is -1 (or missing). Return value is zero if the operation succeeded, negative if it failed.

- `record.hdf5FileRead(file_id, entry=-1)`
  Perform a random-access read into this HDDM record from an input HDF5 file identified by file_id .The record is read from the output dataset at offset entry, or one past the last written record if entry is -1 (or missing). Return value is zero if the operation succeeded, negative if it failed.

- `hddm_*.hdf5FileStamp(file_id, [user_tags])`
  Write the HDDM metadata stamp to the output HDF5 file identified by file_id, optionally supplemented by any number of user-defined tag strings that may be useful to help identify the dataset later when it is read. This method is implicitly called by hdf5FileCreate. Return value is zero if the operation succeeded, negative if it failed.

- `hddm_*.hdf5FileCheck(file_id, [user_tags])`
  Verify the HDDM metadata stamp in the HDF5 file identified by file_id, optionally supplemented by any number of user-defined tag strings to help identify the dataset. This method is implicitly called by hdf5FileOpen. Return value is zero if the operation succeeded, negative if it failed.

- `hddm_*.hdf5DocumentString(file_id)`
  Read the HDDM document string from the HDF5 hdf5 file identified by file_id. The file may be open either for reading or writing. The return value is the HDDM metadata stamp that is stored in the file.

- `hddm_*.hdf5GetEntries(file_id)`
  Returns the number of records currently stored in the HDDM dataset in the HDF5 file identified by file_id, or a negative value if the request failed.

- `hddm_*.hdf5GetChunksize(file_id)`
  Returns the HDF5 chunksize on a HDDM dataset stored in an open HDF5 file identified by file_id, or a negative value if the operation failed.

- `hddm_*.hdf5SetChunksize(file_id, chunksize)`
  Sets the HDF5 dataset chunk size to be used when writing a HDDM dataset to the HDF5 file identified by file_id. This operation only succeeds if it is called between the time when file_id was created and when the HDDM dataset was first written to it. Return value is zero if the operation succeeded, and negative if it failed.

- `hddm_*.hdf5SetFilters(file_id, filters)`
  Sets the list of filters active on the output HDF5 file identified by file_id. The argument filters should be a mutable python list. It is preserved by this function. Filters must be set after file_id is opened but before the first dataset record is written, otherwise they are ignored. Setting filters on a file_id opened for input is an error. Return value is zero if the operation succeeded, and negative if it failed. The following filter identifiers are defined constants in the python module for convenience, but any valid filter identifier may be included in the list.

  - `k_hdf5_gzip_filter` - gzip standard compression provided by hdf5
  - `k_hdf5_szip_filter` - szip standard compression provided by hdf5
  - `k_hdf5_bzip2_plugin` - bzip2 lossless compression used by PyTables
  - `k_hdf5_blosc_plugin` - Blosc lossless compression used by PyTables
  - `k_hdf5_bshuf_plugin` - bitshuffle shuffle filter at bit level instead of byte level
  - `k_hdf5_jpeg_plugin` - JPEG-XR compression filter used in jpeg images
  - `k_hdf5_lz4_plugin` - LZ4 fast lossless compression algorithm
  - `k_hdf5_lzf_plugin` - LZF fast lossless compression used by H5Py project
  - `k_hdf5_lzma_plugin` - modified LZMA compression filter (MAFISC)
  - `k_hdf5_zfp_plugin` - zfp rate, accuracy, or precision bounded compression for arrays of floats

- `hddm_*.hdf5GetFilters(file_id, filters)`
  Gets the list of filters active on the HDF5 file identified by file_id. The argument filters should be a mutable list. It is overwritten by this function. Return value is zero if the operation succeeded, and negative if it failed. The following filter identifiers are defined constants in the python module for convenience, but any valid filter identifier may be included in the list.

C. thread safety with HDF5

The HDDM i/o library is thread-safe, meaning that it is permitted to read or write simultaneously from multiple threads within a single application without interlocks. Contention for access to the underlying data source or sink is taken care of automatically by the library. In fact, users are encouraged to use multithreading to do overlapping i/o operations in order to take advantage of parallelism in the compression/decompression code.

The HDF5 library advertises itself as thread-safe. No additional reentrancy protections have been added to the HDDM i/o interface for HDF5 beyond what is provided by the HDF5 library itself. Some i/o operations only make sense if they are serialized, for example creating a new output file, setting the chunksize on an output dataset, or assigning filters. The first read or write to a newly opened HDF5 file or group has to perform some initial setup operations on the dataset, so the first read/write on a file should be protected in the user code against overlap with subsequent read/write operations.

In the random access of HDDM records in a dataset, there is a possible race condition implied by the fact that reads and writes access the records in sequential order by default, unless an explicit offset is requested by passing it in the entry argument. A multi-threaded application can avoid any risk of race conditions by doing an initial dummy hdf5GetEntries call to trigger the dataset setup when the file is first opened, and then specifying the desired entry argument on any reads or writes requested within a worker thread.

Acknowledgments

This work is supported by the U.S. National Science Foundation under grant 1812415

---

[1] "Representing Scientific Data on the Grid with BinX, Binary XML Description Language", M. Westhead and M. Bull, University of Edinburgh, January 2003. https://www.researchgate.net/publication/238307424_Representing_scientific_data_on_the_Grid_with_BinX-binary_XML_description_language

[2] "RFC 1832 (rfc1832) - XDR: External Data Representation standard", September 1995. https://tools.ietf.org/html/rfc1832

[3] "HDF5 Users's Guide", HDF5 Release 1.10, June 2019. https://portal.hdfgroup.org/display/HDF5/HDF5+User+Guides