

# 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 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.

## Contents

<b>I. Introduction</b>	2
<b>II. XML schemas and HDDM templates</b>	2
A. rules for constructing HDDM templates	3
B. rules for constructing HDDM schemas	4
C. class relationships and model evolution	5
<b>III. Overview of the HDDM toolkit</b>	6
<b>IV. HDDM in python</b>	7
A. writing HDDM files in python	7
B. reading HDDM files in python	8
C. python API reference	9
<b>V. HDDM in C++</b>	9
A. writing HDDM files in C++	10
B. reading HDDM files in C++	11
C. C++ API reference	12
<b>VI. HDDM in c</b>	12
A. writing HDDM files in c	13
B. reading HDDM files in c	15
C. c API reference	17
<b>VII. Advanced features</b>	17
A. on-the-fly compression/decompression	17
B. on-the-fly data integrity checks	18
C. random access to HDDM records	19
<b>VIII. Acknowledgments</b>	19
<b>IX. References</b>	20
<b>References</b>	20

```

<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.xml` 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. In some ways it is like sample data that the programmer might provide to a user to demonstrate how to use it, although the comparison is not perfect.
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 group of schemas that share a basic set of tags. 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 and ignored.
5. An element may have two 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 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. One way a template is not like sample data is that it does not contain actual numerical or symbolic values for the fields in the structure. In the place of actual values, the types of the fields are given. 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 other cases are obvious.
10. Attributes in the template that do not belong to this list are assumed to be constants. Constants are sometimes useful for annotating the xml record. They must have the same value for all instances of the element 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 be repeated more than once for an actual sample. One obvious example of this is a physics event, which is represented only once in the template, but repeated multiple times in a file.
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 inserted in the element 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 allowing users to include arrays as a simple space separated string of values, but the chosen method 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 descendent of itself. Such recursion is complicated to handle and it is hard to think of a situation where it is necessary.
16. Examples of valid hddm templates are given in the examples section below.
17. Because templates contain new tags that are invented by the programmer, it is not possible to write a standard template schema against which a programmer can check his new xml file for use as a template. Instead of using schema validation, the programmer can 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 xml file that must be fixed before it can be used as a template.

## 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 schema is a valid HDDM schema, however, because xml allows for several different ways to express a given data structure.
2. GlueX programmers are not obligated to write schemas that conform to the HDDM specification, but if they do, they have the help of the HDDM tools for efficient file storage and i/o.
3. 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 extensions introduced for the specific needs of GlueX are assigned to a private namespace called "<http://www.gluex.org/hddm>" that is denoted by the prefix *hddm:* .
4. The top element defined by the schema must be `<hddm: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. See below for more details.
5. The names of elements below the root `<hddm:HDDM>` element are user-defined, but they must be constructed according to the following rules.
6. An element may have two kinds of content: child elements and attributes, and hence must have *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. Examples are shown below.
7. 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.
8. All elements in the model document either hold attributes, contain other elements, or both. Empty nodes are meaningless, and are not allowed.

9. Text content between open and close tags is allowed in documents (*type="mixed"*) but it is treated as a comment and stripped on translation. Basic HDDM schemas do not use *type="mixed"* elements.
10. 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.
11. 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.
12. A datum that is sometimes absent can be expressed in the model by assigning it as an attribute to its own host element and putting the host element into its parent with *minOccurs="0"*.
13. 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 comments 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.
14. 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.
15. 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 like is 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 choose the better model, although of course they could just mimic the habitual use of lists by filling the data tree with long strings of monads!
16. 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.
17. 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.
18. 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.
19. An element may be used more than once in the model, but it may never appear as a descendent of itself. Such recursion is complicated to handle and it is hard to think of a situation where it is necessary.
20. 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 GlueX 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** 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, however 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 using the *hddm-c* tool for its i/o interface 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 c-structures corresponding to the missing elements will be empty, i.e. zeroed out.
8. The c/c++ i/o library rejects 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`, 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 `addXXXs()` 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 `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. 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

See section VII Advanced features below.

## 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 `addXXXs()` 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 `addXXXs()` 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

See section VII Advanced features below.

## 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.glutex.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 `addXXXs()` 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 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. ACKNOWLEDGMENTS

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

## IX. REFERENCES

---

- [1] "Representing Scientific Data on the Grid with BinX, Binary XML Description Language", M. Westhead and M. Bull, University of Edinburgh, January 2003. "<http://www.epcc.ed.ac.uk/gridserve/WP5/Binx/sci-data-with-binx.pdf>"
- [2] "'HDF5 Users's Guide'", working draft, July 2003. <http://hdf.ncsa.uiuc.edu/HDF5/doc/UG/>
- [3] "'RFC 1832 (rfc1832) - XDR: External Data Representation standard'", September 1995. <http://www.faqs.org/rfcs/rfc1832.html>