HDDM Programmer's Interface

From GlueXWiki

Contents

Introduction

HDDM was introduced in the context of GlueX as a means to encode output from Monte Carlo simulations and results from their reconstruction. To understand why we needed something like HDDM, rather than going with a community-based standard such as HDF, see [1] below. That reference also contains a description of the design principles and requirements for the software package. The purpose of this wiki page is to give a quick-start guide for programmers that might want a way to write new hddm files or read data from existing files. The package comes with a set of tools and programmer interfaces that makes this very easy to do, particularly with python. The underlying implementation is in C++, so it provides good performance in terms of data rate to/from disk files with serial access. On-the-fly compression/decompression and automatic integrity verification are built into the package. Random-access to events at any location in a file without reading the entire file is also supported.

Templates and schemas

HDDM files are built from an xml template. A hddm template is a short xml document that describes the structure of one record in the hddm stream. Every hddm file has a copy of its template at the beginning, followed by its event data in a compact binary format. The template is what arranges those data into a meaningful structure. A simple example of a template is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<HDDM class="x" version="1.0" xmlns="http://www.gluex.org/hddm">
  <student name="string" minOccurs="0">
    <enrolled year="int" semester="int" maxOccurs="unbounded">
      <course credits="int" title="string" maxOccurs="unbounded">
        <result grade="string" Pass="boolean" />
      </course>
    </enrolled>
  </student>
</HDDM>
```

All of the events 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 are treated as literal strings, 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. Others like minOccurs/maxOccurs tell the data model whether a given element is always present in every event or may be omitted (minOccurs="0" indicates this) or whether it may be repeated any number of times (maxOccurs="unbounded" indicates this). The top-level element is special in that it must always be named HDDM and have the attributes shown above. The class attribute is an abbreviation that you chose for the data model you are creating. Chose a short, unique name for your class because it is used in filenames that are written by the hddm tools, and the abbreviation prevents collisions between files built from different templates (classes).

Templates provide an intuitive informal way of specifying the structure of a record in a hddm file. For most users, this is all they need to know about, but for those familiar with XML 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 above template conforms to a schema called "http://www.gluex.org/hddm" (hint: this is not a URL to anywhere, it is a URI known as an *xml namespace*), as indicated in the xmlns attribute in the HDDM tag of the template. The schema for this document type is found in hddm_schema.xsl in the main hddm directory of the distribution. The second use of schemas is that every hddm file is itself a valid xml document, so it needs a schema against which it can be verified. 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 much 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 very limited, except for specialists. For the remainder of this document, we will deal only with templates.

How to get started

The hddm toolkit is distributed as a part of the GlueX sim-recon distribution. The sim-recon distribution is distributed from the github repository as JeffersonLab/sim-recon. Instructions for how to download and build sim-recon are given elsewhere on this wiki. The hddm tools are located in sim-recon/src/programs/Utilities/hddm. Checking out the repository, setting up your build environment, and executing "scons -u install" from sim-recon/src/Utilities/hddm should be all that is needed to build the hddm toolkit. Before continuing to read this document, make sure that the basic tools like 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 tools you need to build the libraries from a template.

Before you can begin to work with hddm files, you need a template. There is a template at the head of every hddm file, so if you have a hddm file that has already been created that you want to work with, simply extract the header using a text editor and save it to a file with the extension ".xml". Another way to get started would be to copy/paste the above example template into a file "exam2.xml" (or copy it from the distribution directory). The instructions that follow assume that you have done this. Now it is time to build a hddm i/o library to let you read and write hddm data records. Currently there are 3 programming languages supported by hddm: python, C++, and c. Python is the least verbose and most readable interface, so let's begin with that.

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 file>] <input hddm file>
```

The reverse action is provided by the xml-hddm tool, assuming that the user has a copy of the template and the input data file already formatted as a valid xml file.

```
$ xml-hddm -t <template> [-o <output hddm file>] <input xml file>
```

Since the full xml rendition of a data file with many records is extremely verbose, this tool is of limited use in actual practice, except to process the output from the hddm-xml file and run the decoding procedures in reverse. This 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 keep track of what a hddm file actually contains. Keep them handy when working through the language-specific procedures below.

HDDM in python

If you have access to a hddm file that was written by someone else, copy it into your work directory and use a text editor to extract the header into a file, which you may call "exam.xml". Use the following commands to build the python module that you will need to read the contents of this file.

```
$ hddm-cpp exam.xml
$ hddm-py exam.xml
$ python setup_hddm_X.py build -b build_hddm_X
```

In this example, I assumed that the HDDM "class" letter (see the HDDM tag in your template header) was "X". You should change it to whatever the actual class abbreviation is for your hddm file. The above steps should create a shared library that starts with hddm_X in your work directory. Copy that module to a place in your PYTHONPATH where you usually place your private python modules, then execute the following program to print the contents of your hddm file in plain text. I assumed it was called "exam.hddm".

```
import hddm_X
for rec in hddm_X.istream("exam.hddm"):
   print rec
```

To see the same data printed out as a properly formatted xml document, replace the "print rec" with "print rec.toXML()".

writing hddm files in python

For this example, I return to the template listed at the top of this page, which I call "exam2.xml". Using the build steps above, build the python module hddm_x, then try the following code to write a new output hddm file from scratch, starting only from the template.

```
import hddm_x
ofs = hddm_x.ostream(“exam2.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)
```

Copy this python program to a file and execute it using the python interpreter. This generates a new hddm file called exam2.hddm. Now running the above 3-line python print program on exam2.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 structure of the output record you are writing is already known to the program because it knows about your template. All that you need to do is to fill in the elements and assign the values of the attributes. You begin by creating an empty record by calling the HDDM() default constructor. Then you populate the structure top-down by calling addXXXs() methods for each tag XXX under that. The name XXXs is the name of the tag element in the template in 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 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, although xml rules require that you specify minOccurs="0" for the containing tag in the template if you plan to do that. 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.

reading hddm files in python

For this illustration, I assume you have created the file exam2.hddm using the instructions 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 student.

```
import hddm_x
ifs = hddm_x.istream("exam2.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
print course.name, "enrolled in", total_courses, " courses", \
       "and passed" , total_passed, "of them,\n",\
       "earning a total of", total_credits, "credits.\n"
```

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, listed 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("exam2.hddm"):" would look over all records in the input file with the rec loop variable being the HDDM element from each record in the input file. Likewise, each call to method getXXXs() returns a python list of tag element objects that is iterable using "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 list functions (eg. len(list), str(list), repr(list)) all work as expected for these tag list objects returned by getXXXs() methods. This interface was designed to be pythonic, ie. "there should be only one (obvious) way to do it." so most things should work intuitively. It is especially powerful when combined with pyroot to allow a quick-and-simple prototyping framework for physics analysis.

advanced features of the python API

See section on Advanced features below.

HDDM in C++

If you have access to a hddm file that was written by someone else, copy it into your work directory and use a text editor to extract the header into a file, which you may call "exam.xml". Use the following commands to build the C++ module that you will need to read the contents of this file.

```
$ hddm-cpp exam.xml
$ mv hddm_x.cpp hddm_x++.cpp
$ g++ -c  hddm_x++.cpp XString.cpp XParsers.cpp md5.c -I $HALLD_HOME/$BMS_OSNAME/include \
-I$XERCESCROOT/include -L $XERCESCROOT/lib -l xerces-c -L $HALLD_HOME/$BMS_OSNAME/lib \
-lxstream -lz -lbz2
```

The rename step from hddm_x.cpp to hddm_x++.cpp is inserted to prevent any confusion between the files created in this section and those created below for use with the c API, but is not essential if this is the only interface that is of interest to you.

writing hddm files in C++

For this example, I return to the template listed at the top of this page, which I call "exam2.xml". Use the build steps above to build the hddm_x C++ API object module. Create a new file 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(“exam2.hddm”);
   hddm_x::ostream ostr(ofs);
   ostr << xrec;
   xrec.clear();
   return 0;
}
```

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

```
$ g++ -o write_exam write_exam.cpp hddm_x++.o -I. -I $HALLD_HOME/$BMS_OSNAME/include \
-I$XERCESCROOT/include -L $XERCESCROOT/lib -l xerces-c -L $HALLD_HOME/$BMS_OSNAME/lib \
-lxstream -lz -lbz2
```

This may need to be customized for your own build environment. Once it completes successfully, you will find the executable write_exam 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.

```
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 structure of the output record you are writing is already known to the program because it knows about your template. All that you need to do is to fill in the elements and assign the values of the attributes. You begin by creating an empty record by calling the HDDM() default constructor. Then you populate the structure top-down by calling addXXXs() methods for each tag XXX under that. The name XXXs is the name of the tag element in the template in 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 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 containing tag in the template if you plan to do that. As soon as a new element list is created, you can fill in the values of its 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.

reading hddm files in C++

For this illustration, I assume you have created the file exam2.hddm using the instructions in the previous section. The following C++ 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 student.

```
#include <fstream>
#include "hddm_x.hpp"
int main()
{
   std::ifstream ifs("exam2.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.
```

The HDDM tag list objects are std:list containers so they support STL forward iterator semantics. 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, listed in the order of their appearance. Each call to a method getXXXs() returns a C++ list of tag element objects. The individual attributes of each tag instance are accessed using the get<Attname> members of their host object. The standard list methods (eg. size(), begin(), end()) all work as expected for these tag list objects returned by getXXXs() methods. If you are not sure how to do something, a quick browse through the header file should give a good overview of the capabilities of these classes.

advanced features of the C++ API

See section on Advanced features below.

HDDM in c

If you have access to a hddm file that was written by someone else, copy it into your work directory and use a text editor to extract the template header into a file, which you may call "exam.xml". Use the following commands to build the c library that you will need to read the contents of this file.

```
$ hddm-c exam.xml
$ g++ -c  hddm_x.c XString.cpp XParsers.cpp md5.c -I $HALLD_HOME/$BMS_OSNAME/include \
-I$XERCESCROOT/include -L $XERCESCROOT/lib -l xerces-c
```

writing hddm files in c

For this example, I return to the template listed at the top of this page, which I call "exam2.xml". Use the build steps above to build the hddm_x c API object module. Create a new file and cut/paste the contents of the box 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;
}
```

Copy 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 $HALLD_HOME/$BMS_OSNAME/include \
-I$XERCESCROOT/include -L $XERCESCROOT/lib -l xerces-c
```

This may need to be customized for your own build 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.

```
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+"
```

This example explains most of what you need to know to set up hddm structures in memory, and then 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 himself. 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 to hold it. After that, the memory is owned by the top-level HDDM struct, 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 step. Either way, flush_x_HDDM() frees all memory owned by the HDDM record, discarding its contents, before it returns.

The structure of the output record you are writing is already known to the program because it knows about your template. All that you need to do is to fill in the elements and assign the values of the attributes. You begin by creating an empty record by calling make_x_HDDM(). Then you populate 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. The name XXXs is the name of the tag element in the template in 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 event. This is a perfectly valid hddm record, but it must be checked for by the code that reads the record since c has no automatic checking of the validity of pointers during dereferencing. 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.

reading hddm files in c

For this illustration, I assume you have created the file exam2.hddm using the instructions in the previous section. The following c 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 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 event is read in. Accessing leaf elements that are deep inside the hddm template hierarchy can only be reached by traversing all of the nodes in the tree above, which makes a simple data mining operation somewhat verbose, although it still scales well because of the hierarchical model (not like a linked list). If you are unsure about how to do something, browsing through the header file is probably not going to be easy because all of the internal functionality of the logic that supports the serialization/deserialization of the data is exposed there. The API is fairly simple: Access to the attributes themselves 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 for all of the other API's, the template itself should be the main documentation needed to write code that interacts with hddm files.

advanced features of the c API

The c API is no longer in active development, so it is supported only for legacy applications that rely on it. The features described in the Advanced features section below are not available using the c API. The only things that are ensured by the ongoing support of the c API is that it can files based on any valid hddm template, and that hddm files written using it can be read by applications built using any of the other API's. The converse is not true. If an input file is not readable by the c-API, it prints a polite error message reporting this fact and quietly exits.

Advanced features

on-the-fly compression/decompresson

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 file that tells the reader when to enable/disable decompression on the input stream. These transitions occur silently during input; no user action is needed. Two compression options are supported.

1. **bzip2 compression** - This option offers the best compression ratio, a factor of about 2.5 for GlueX 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 GlueX Monte Carlo data, but 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. 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 events in the stream (for more about random access, see below). If the stream is uncompressed, random access to a particular event record generates a read starting at the beginning of that specific record and only taking in the contents of that record. If the stream is compressed, the entire compression block containing the event 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 choice between these three options that will be best in every situation. The one producing the data should consider what the most likely scenarios are for reading the data, and weight the costs and benefits of compression before simply opting for bzip2.

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 values should be passed as mode to the setCompression(mode) method of the hddm_x::ostream class to change the current state of the output stream. Only records written after this method is called will reflect this 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.

The python istream and ostream objects support the exact same interface. The named constants listed above are defined within the hddm_x module namespace.

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 function 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 that it matches what the same hash algorithm returns when applied to the data record read 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 be reflected in a different value for this 32-bit code, and it is very difficult to get errors that cancel out and generate the same crc. This 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. I suppose one might consider this marginally better for error detection, but it is more expensive to compute. Neither MD5 nor CRC32 options result in any noticeable overhead in the context of standard GlueX data streams.

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. Only 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 modes is presently enabled.

The python istream and ostream objects support the exact same interface. The named constants listed above are defined within the hddm_x module namespace.

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 after the end of the previous output record. However, the stream position pointer of the stream right before any given record was written can be recorded by the process writing the data, and then provided later to a read process which might use it to seek out particular records of interest and read only those. 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.

Stream positions can only be recorded/set at the start of event records. Attempts to access a stream at a random position, or at a position that was generated by another hddm stream, will result in unpredictable behavior, most likely a segmentation fault. 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 event
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 these values, a minimum field width of 128 bits would be needed. Of course the name and creation date of the input file that contains this record is needed the complete the information needed to look up this event. 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 individual instances of that value are not generally needed for a list of record positions.

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, simple call method getPosition() within a try-catch block and catch any RuntimeError that is thrown as indicating that the file 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 methods called getPosition() which return objects of type hddm_x.streamposition, which can be passed back through hddm_x.setPosition() to seek a new read position in the input stream. Unless another setPosition() is executed, reading proceeds in a serial fashion starting from the last event read from the stream.

References

[1] HDDM "Hall D Data Model", R.T. Jones, Gluex-doc-065, September 23, 2003. (http://argus.phys.uregina.ca/cgi-bin/private/DocDB/ShowDocument?docid=65)