MDSL

MINI DATA STRUCTURE LANGUAGE

(version 1.0)

SPECIFICATION, TUTORIAL & LIBRARY REFERENCE

CARLOS EDUARDO MELLO
Universidade de Brasília

February 2016

Núcleo de Computação Musical (NCM/UnB)
Departamento de Música, Universidade de Brasília

Thank you for your interest in MDSL. I hope you find it useful and easy to use. If you end up including it in your own projects please drop me a line at melloedu@unb.br. If by any chance you are interested in collaborating with the project, please refer to GitHub. If you just want to try it, simply download its zipped folder from the project's page and have fun with it.

Carlos Eduardo Mello

UnB, February 18 2016

INTRODUCTION

In very simple terms, the technology know as **Extensible Markup Language, or XML,** uses tags to organize information in text-based computer files. For those who are used to writing web pages, XML feels like HTML, except that HTML tags are used specifically for formatting displayed text and images, while XML is more concerned with the structure of data and the meaning of the tags is open to be defined by the application using it. **MDSL** (**Mini Data Structure Language**) is a simple XML-based language which supports a very small subset of XML features. It is suitable for creating easy-to-use, text-based file formats, for storing and retrieving structured information in software with simple data storage requirements.

Tags are Fun!

All right but we are talking about data files... Why would anyone want to use a text-based file format for storing data instead of a binary format? There are many reasons, but the most compelling one is probably *readability*. Is binary a lot more efficient? Yes. Does it take a lot less space? Definitely. Then why should I use 5 bytes to store a value like 65535 when I can use only 2? Because it is a whole lot easier to read! But, wait a minute... isn't the computer going to do all the reading, so why should I care? Because as a programmer you'll have a much easier time editing, testing, revising your code if you can look at the data set in your file with your own eyes. For example, suppose you've been tracking a bug on your code for hours and you want to check your output file trying to figure out what is going on. Let's say the data is a set of numbers resulting from some calculations in your code. Which one of the following would you rather look at?

`%&*~`f@..!` or `9510 10878 24678 16430 11809`

OK, but why use tags? Why add all that verbose overhead instead of just the data itself? Because tags are fun! They are a simple and quick way to add structure to your information, yet still keeping it easy to read. Also, tags help you maintain your software. It is perfectly possible to store a bunch of data in a file, in any arbitrary sequence/order and design your code so that each piece of data is always stored and retrieved intact. However, if later on you need to change that order by any reason, you may have to rewrite your file accessing code entirely. Using tags you can track where each piece of data is in your file and find it safely even if you move it around the file, as long as the tag structure is preserved. It is much like when you use a `struct` in your code. You don't need to know exactly where each field is placed in the `struct`'s memory block, because using the access operator you always reach the correct piece of information.

Tags are markers that delimit a part of the structure. They come in pairs, one for opening and one for closing. Anything placed between a pair of tags is considered part of the element the tags are delimiting. An opening tag comes enclosed in '<' and '>' characters and the closing tag is just like the opening one except for an added slash after the '<' character, like this:

```
<B>Text</B>
```

There are a lot of languages that use tags for formatting text, HTML being perhaps the most popular one. As you may already know, in HTML all possible tags are already defined and are read by a hypertext interpreter, like a web browser, according to specific rules. In the text above, for example, the `<B></B>` is used to indicate text that should be displayed in boldface. The fun of XML-based languages is that you can invent any tags that suit your needs and give them any meaning you want, according to your program design. As a subset of XML, MDSL is all about structuring your data with tags. You decide which structure you need for your data and define the tags you need to enforce that structure. Than you use the MDSL Library to read and write that data to/from a file.

Suppose you were writing an application for keeping track of your cd collection. You could define an element called cd and insert information like the cd's title, artist and recording date within that element, like this:

```
<cd>Pink Floyd. The Wall, 1979</cd>
```

However, if you added further structure to your element, it would be easier to retrieve each information without having to parse the single string every time. Each piece of information you want to store about a cd could be formatted as an element inside the cd element. Your collection of cds could then be represented by a sequence of cd elements within an enclosing tag, like this:

```
<cd_collection>
        <cd>
                <title>The Wall</title>
                <artist>Pink Floyd</artist>
                <year>1979</year>
        </cd>
        <cd>
                <title>Beethoven's 9 Symphonies</title>
                <artist>Herbert von Karajan</artist>
                <year>1963</year>
        </cd>
</cd_collection>
```

This is the kind of data file you'd be creating with MDSL. A simple tree of XML-like elements that you can put together easily and parse with help from MDSL's Library. In general, building this kind of output when writing text files is a fairly straight forward task. In fact you can, and probably should, write your own routines for this. As you will be able to see by reading this manual, the real advantage of using MDSL happens when reading the file.

MDSL SPECIFICATION

The **MDSL** format, discussed in this document, is based on the XML specification. Basic understanding of XML is recommended and will help the reader assimilate the information in this manual. Suggested sources of information on the subject are:

- http://www.xml.org
- Harold , E. & S. Means. *XML in a Nutshell*. O'Reilly: 2001 (ISBN: 0-596-00058-8)

DOCUMENT FORMAT

The basic design of **MDSL** is very simple. Starting from XML fundamental structures (markup tags) it adds a few restrictions and discards many features, in order to keep things as simple as possible. The rules bellow show how **MDSL** differs from the XML standard.

1. Every **MDSL** document should be a well-formed XML document. However, not every XML document qualifies for use with **MDSL**.

2. Only **one type of tag** is permitted in MDSL documents: the **element tag**. Opening and closing tags must be present and correctly balanced for every element. Data structure must be constructed by embedding elements within other elements in order to describe a document hierarchy (obs.: the XML declaration tag is currently not allowed).

3. Abbreviated tags (e.g.: <element/>) are not allowed in **MDSL**.

4. Each element may contain either **character data** OR **other elements** between its enclosing tags, but **not both**. This way, the actual data will be contained in the bottom level nodes of the hierarchy tree. When an element contains other elements, the number of enclosed elements can be either free (i.e.: as many as needed) or only one (unique), as specified by the associated **Structure List**.

5. Each element definition must be unique within the entire document. In other words there cannot exist two elements with the same name and different definitions. The term definition here is understood as the exact structure an element carries within.

6. Element attributes are not allowed. Neither are any other XML constructs. In order to build a hierarchy that reflects specific data structures with **MDSL**, you must create a tag for each piece of data and embed it in the corresponding element, in place of the intended attribute. For example, instead of using `<actor first_name="John" last_name="Wayne"/>`, you need to use:

```
<actor>
      <first_name>John</first_name>
      <last_name>Wayne</last_name>
</actor>
```

otherwise the document will be discarded as unreadable. Although this scheme has obvious limitations, it fulfills the needs of many simple applications. Also, it makes parsing and other processing a lot simpler while retaining file readability.

STRUCTURE LISTS

MDSL uses a strategy for document structure checking which is analogous to that of *DTDs* and validating parsers. However, besides checking for data structure, **MDSL** also verifies data types and validity. This is accomplished by comparing the data tree with a **structure list,** a short but complete list of valid elements along with their basic data types (text, integers, floats, etc.), number of similar elements allowed in a given node level, and so forth. This structure/data checking is done during parsing, so that once that step is completed without errors, the user has a clean data set which can be retrieved with very little effort.

Structure lists are associated with specific applications which dictate its structure and use them to control their files. The structure of a given **MDSL** document is defined by its main element, which must be a unique tag that happens at the top level of the hierarchy. When a document is parsed, this tag is searched for. If found, its structure is analyzed and tested. Everything outside this main tag is ignored. Therefore, it is possible to embed an **MDSL** document within any text file, as long as its inner integrity is intact. On the other hand, if the main element for a given application document structure is not found, the file is deemed useless and discarded as invalid.

Note: The correct path for the **Structure List** file must be provided at the time of parsing!

It should be noted that validating a file with a Structure List is optional. The MDSL API makes it possible to bypass this verification. However, if structure validation is not performed, a malformed document could produce all sorts of parsing errors, data corruption and even disrupt program execution. So it is generally a good idea to create a valid structure list to be used when opening files for each specific MDSL application.

For more details on how to configure a structure list to use with a specific application, see the **Structure Lists** section in **Library Reference**, later in this document.

In terms of its specification, **MDSL** is compatible with the XML language. Documents created using its format can be read by an XML parser. However, instead of relying on feature-rich parsers which implement every aspect of the XML specification, MDSL comes with its own management library, which handles everything from parsing or constructing a data tree, to performing structural checking and retrieving or storing information. All this is done through a compact, user-friendly API set.

The main idea behind the **MDSL Library** is to provide a set of methods that will retrieve data associated with specific tags directly to user code. Every intermediary step of the process, such as traversing the data tree and doing other house keeping, is handled transparently by the library. The MDSL Library is built with Standard C++ (ANSI). The library is distributed as source code (currently maintained as an open source project out of GitHub). The library consists of a single class of objects, the `MDSLControl` class. `MDSLControl` is implemented in a pair of files (`MDSControl.h and MDSLControl.cpp`) which can easily be included, compiled and used by a C++ project. The code makes no use of any platform specific resources. All i/o is handled with C++ stream facilities and should work seamleslly across all platforms with an ANSII C++ compiler.

The remainder of this document consists of a **Tutorial** which demonstrates MDSL's main features and walks the user through its typical workflow. The tutorial goes into detail through expected usage scenarios, so that library code can be put to use right away. If nothing else, it is a good idea to look through the code examples in the Tutorial. The source code for the tutorial consists of a `main.cpp` file, which is available at GitHub along with the library's classes and a few other supporting files.

At the end of the Tutorial there is a complete **Library Reference** section which documents every available API for MDSL. There are also a couple of appendices which provide further details about the specification.

TUTORIAL

CREATING MDSL FILES

The whole advantage of using a language like MDSL is to be able to store your application's data so that it can be retrieved easily and kept tidy without spending too much time writing file access code. The first step in that direction is designing the document format.

Applications commonly utilize things like a `struct` or object to group data pieces that belong together, then declare arrays or lists of these units to group data in memory. Translating that kind of arrangement to an MDSL data structure feels rather natural. For example, if an application handles personal contact information for an address book, it may manage that information in memory using `structs` as in **Example 1a**. This model would lend itself very naturally to something like **Example 1b** for file storage of the same kind of data.

a) data in memory:

```
struct date_of_birth{
       short day;
       short month;
       short year; };

struct contact{
       string first_name;
       string last_name;
       string email;
       date_of_birth dob; };
```

b) data on file:

```
<addressbook>
       <contact>
              <firstname>Steven</firstname>
              <lastname>Jobs</lastname>
              <dob>
                     <day>24</day>
                     <month>2</month>
                     <year>1955</year>
              </dob>
       </contact>
<contact>
              <firstname>William</firstname>
              <lastname>Gates</lastname>
              <dob>
                     <day>28</day>
                     <month>10</month>
                     <year>1955</year>
              </dob>
       </contact>
</addressbook>
```

Example 1 - data structures and element tags

Writing the file directly

One way of generating the file structure suggested above would be to output each piece of data from the structures in memory surrounding it with the appropriate tags. A function like the one in ***Example 2*** would handle the job just fine. The function receives a string with the file address, a pointer to an array of contact structures and the number of items in the array.

```
void WriteToFile(string filePath, contact * contacts, int numberOfContacts)
{
    ofstream outputFile;

    outputFile.open(filePath.c_str());
    if(outputFile.is_open())
    {
       outputFile << "<addressbook>" << endl;

       for(int i = 0; i < numberOfContacts; i++)
       {
              outputFile << "<contact>" << endl;

              outputFile << "<firstname>";
              outputFile << contacts[i].first_name;
              outputFile << "</firstname>" << endl;
              outputFile << "<lastname>";
              outputFile << contacts[i].last_name;
              outputFile << "</lastname>" << endl;
              outputFile << "<email>";
              outputFile << contacts[i].email;
              outputFile << "</email>" << endl;
              outputFile << "<dob>" << endl;
              outputFile << "<day>";
              outputFile << contacts[i].dob.day;
              outputFile << "</day>" << endl;
              outputFile << "<month>";
              outputFile << contacts[i].dob.month;
              outputFile << "</month>" << endl;
              outputFile << "<year>";
              outputFile << contacts[i].dob.year;
              outputFile << "</year>" << endl;
              outputFile << "</dob>" << endl;

              outputFile << "</contact>" << endl << endl;
       }
       outputFile << "</addressbook>" << endl;
       outputFile.close();
    }
}
```

Example 2 - Writing MDSL data standard io

This is a simple but effective way of generating the desired output file. The advantage of using a simple routine like this is that it can be quickly adapted to whatever data structure you may have. Generally speaking this approach is preferred for simplicity as well as for performance reasons, as we will discuss further bellow.

Using the Library for file generation

The other possibility for building an MDSL document from scratch is to use the Library. For this you need to create the main element and add other elements according to your application's design. After all the data is entered, you can write the tree to a file with a single call to `MDSLControl::WriteToFile()`. In order to demonstrate this process, lets consider another hypothetical application: a classroom info app. This example will be used with the remainder of the demo code for this tutorial. The complete source code for reading and writing classroom data can be found with the distributed code for the MDSL Library. There you will also find detailed instructions (`READ.ME`) on how to build and test these examples. For now let's follow the idea and try to understand the code.

Data Model

The model for our classroom is illustrated in ***Figure* 1**. The outer box represents the main container in the structure, a *class*, which is our main element. Inside the *class* box we can see a *teacher* box and various *student* boxes. Each one of these sub-boxes represent elements contained in the main *class* element. Following this pattern we find various other elements contained in each box level. All the elements inside a given element are called its children. You may have noticed an element called *name*, composed of two children, *first* and *last*. The *name* element appears inside more than one type of element: *teacher* and *student*. This type of structure is ok as long as every *name* element has the exact same definition (see DOCUMENT FORMAT for more details on MDSL formatting rules).

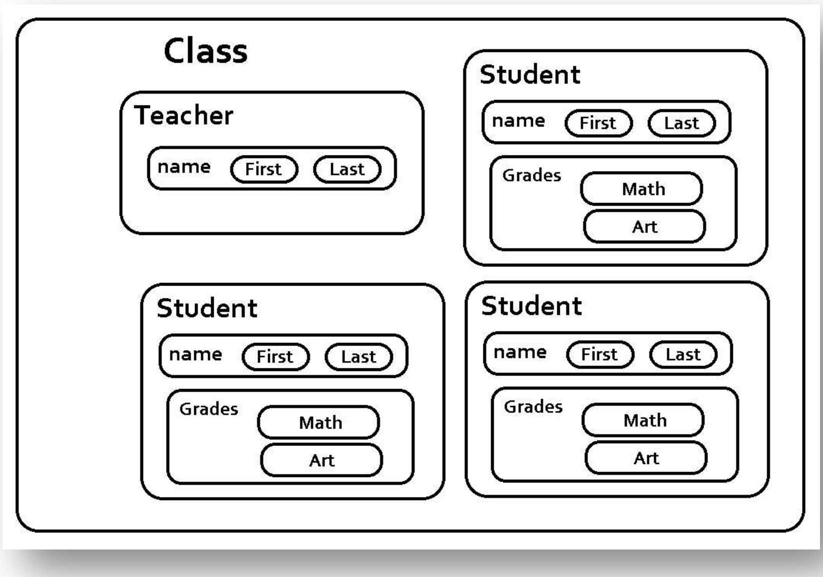

Figure 1 - a classroom representation

The sample code in ***Example* 3** demonstrates how to construct an MDSL file by inserting the appropriate elements in the data tree using an `MDSLControl` object. The very first action you see in the code is a call to `MDSLControl::CreateMainElement()`. This call starts the file structure definition by inserting the initial node in the data tree.

Once you have the main element in place, you add its children, and its children's children, and so on, using `MDSLControl::AddNewElement()`. `AddNewElement()` inserts a child element in the element referred to by its first argument, an element id. The second argument is used as the tag name for the newly inserted element. The third argument, if present, becomes the data string for the element. The other API used throughout this function is `MDSLControl::GetID()`. This method returns the id of a child of the requested element which has the requested name tag. For example, the code adds a *teacher* element under *class*. Then it gets the teacher id by calling `GetID(mainId, "teacher", 0)`. The call is asking for the first child under the main element that has a "teacher" tag. In this case, the element we just added is the only such child. Once the code has a valid id for the teacher element, it can add children to it by calling `AddNewElement()`, and so forth.

```
void GenerateMDSLFile(string fileName)
{
    MDSLControl myControl;
    EP ep = EPzeroERROR;
    long mainId, teatcherId, studentId, nameId, gradesId;
    int count = 0;

    mainId = myControl.CreateMainElement("class");

    ep = myControl.AddNewElement(mainId, "teacher", "");
    teatcherId = myControl.GetID(mainId, "teacher", 0);
    ep = myControl.AddNewElement(teatcherId, "name", "");
    nameId = myControl.GetID(teatcherId, "name", 0);
    ep = myControl.AddNewElement(nameId, "first", "Professor");
    ep = myControl.AddNewElement(nameId, "last", "Wonder");

    while (count < 10)
    {
        ep = myControl.AddNewElement(mainId, "student", "");
        if(ep == EPzeroERROR)
        {
            studentId = myControl.GetID(mainId, "student", count);
            ep = myControl.AddNewElement(studentId, "name", "");
            ep = myControl.AddNewElement(studentId, "grades", "");
            nameId = myControl.GetID(studentId, "name", 0);
            ep = myControl.AddNewElement(nameId, "first", "Excellent");
            ep = myControl.AddNewElement(nameId, "last", "Student");
            gradesId = myControl.GetID(studentId, "grades", 0);
            ep = myControl.AddNewElement(gradesId, "art", "10.0");
            ep = myControl.AddNewElement(gradesId, "math", "10.0");
        }
        count++;
    }
    myControl.WriteToFile(fileName);
}
```

Example 3 - Writing MDSL data with MDSLControl

Data Access

The MDSL Library was designed to be a simple tool for accessing structured data in preferences or small document files. The specific steps required for retrieving each piece of information from an MDSL file will depend on the specific format you design with it, but the basic workflow will usually be similar to the following:

- parsing the file
- getting the main element's id
- accessing other elements starting with the main id

File Parsing

The first step in retrieving data stored in an **MDSL** document is to declare an `MDSLControl` object and ask it to open the file, like this:

```
EP progError = EPzeroERROR;
MDSLControl myControl;
progError = myControl.OpenFile("/home/me/mystuff/myDataFile.mds");
```

Example 4 - Parsing and MDSL file

Example 4 shows how to declare and use **MDSL**'s built-in parser. The `EP` data type is a 16 bit result code used to report possible errors during parsing. `MDSLControl` is the class responsible for every aspect of data access. It contains all the necessary methods to read and write **MDSL** files. It should be declared only once and used throughout an application run. The `MDSLControl::OpenFile()` method opens, parses and checks an **MDSL** file and reports any errors it may encounter along the way. The file is specified with a full pathname, which can be provided through either a standard C++ Library String object or as a C-style string (i.e.: null terminated) or string literal, as in the example. `OpenFile()` returns `EPzeroERROR` if everything works as expected. If any other values are returned, it means the loading operation was aborted and the control should not be used (check Library Reference for possible error values).

Retrieving Element Information

Once the file is loaded with `OpenFile()`, you can access every part of it by calling `MDSLControl`'s retrieving methods. These methods return information about a file structure and its data based on the names of elements you provide when you call them. The top level of the hierarchy tree represents the very first element in the file, which must be unique within the entire file (see DOCUMENT FORMAT). This element is assumed to be the document type for the application and refers to the data as a whole. In other words, the data structure itself starts on its children's level. When `MDSLControl` loads the file, it creates, for each and every element, a unique id number which is retained throughout the element's life. Once the file is loaded, the next step is to get the id for the main element. After that, you can use that number to retrieve the other element id numbers and their respective data. For example, suppose your application is trying to load a classroom info file like the one defined in our data creation example (***Example*** 5):

```
<class>
        <Teacher>
                <name>
                        <first>Nadia</first>
                        <last>Boulanger</last>
                </name>
        </Teacher>
        <Student>
                <name>
                        <first>Aaron</first>
                        <last>Copland</last>
                </name>
                <grades>
                        <math>2.0</math>
                        <art>4.0</art>
                </grades>
        </Student>
        <Student>
                <name>
                        <first>Walter</first>
                        <last>Piston</last>
                </name>
                <grades>
                        <math>2.5</math>
                        <art>3.75</art>
                </grades>
        </Student>
        <Student>
                <name>
                        <first>Roy</first>
                        <last>Harris</last>
                </name>
                <grades>
                        <math>3.0</math>
                        <art>3.5</art>
                </grades>
        </Student>
</class>
```
Example 5 - Sample MDSL file: classroom

In the document above, the top level is represented by the *class* element. Calling `MDSLGetMainID()` will get you the top level id number, in this case, the id for the *class* element. An element id is a 32-Bit long integer generated by `MDSLControl` at the time the file is parsed. Once you have an element's id number you can access its children elements or its data string.

Obs.: It should be noticed that these id numbers are for internal reference and interface calls only. They are not recorded on a file's data structure. If your application needs to number its data structures, you should create ID elements within the elements you need to number, so that these IDs get stored along with other data. Indeed, the whole idea of this access library is to facilitate the interaction of file data and memory resident data. But on the memory side an application should have, and normally does, its own management routines and structures to rely upon.

From this point on, you simply specify which element you need and the index for elements of that type. Since all elements with the same name are identical in structure, it doesn't matter which child element gets loaded first during parsing -- the elements will remain in the tree in the order they were loaded. Thus, to access the *Teacher* element, for example, you would call `MDSLGetID(fatherId,"Teacher",index)`, where `fatherId` is the reference number for the *Teacher* element's father (i.e.: the number returned by the previous call) and *index* is the order number of the element you want to get. To access each Student you would call the same method several times, using the corresponding name tag and each available index -- `MDSLGetID(fatherId,"Student",index)`.

In order to keep track of multiple elements, you should use `MDSLGetNumberOfElements(long elemId)` to know how many children there are on that level, `or` `MDSLGetNumberOfElements(long elemId, string tagName)`, to know how many elements of a given type there are in that element's list of children; in this case, the number of students. With an element's id number you can get its name tag with a call to `MDSLGetTag(long elemId)` or its data string (if present) with a call to `MDSLGetDataString(long elemId)`. Alternatively you may obtain the data converted to either integer or float formats with a call to `MDSLGetDataLong(long elemId) or MDSLGetDataDouble(long elemId)` respectively (see **Library Reference** for the details of number conversion). The code in ***Example 6*** illustrates the situation described above. It should be noted that the calling code must be aware of the format of the data being extracted, in order to make meaningful use of it. Each level of the structure allows access to its children. So, for example, once you have the id of one of the students, you can gain access to all the student's data, such as `name` and `grades`. Once you have the id for that student's name, it's easy to get its parts: `first` and `last`. Following the same steps with the correct tags you can reach every part of the document.

```
#include “MDSControl.h”
#include <iostream>

// The function bellow gets information from the elements and sends it to the standard
// output. It receives two file paths: one for the file to be open and one for the
// structure list for the input file to be validated against.
void OpenMDSLFile(string fileName, string structList)
{
    MDSLControl myControl;
    EP ep = EPzeroERROR;
    long mainId, nameId, gradesId, tempId;
    long  numDeAlunos;

    cout << "Openning Document..." << endl;
    ep = myControl.OpenFile(fileName, structList);
    if(ep != EPzeroERROR)
        cout << endl << "Failed to open document:\t\terror: " << ep << endl << endl;
    else
    {
        cout << endl << "Initiating data extraction..." << endl << endl;

        // the first step is to obtain the main element's id...
        mainId = myControl.GetMainID();
        cout << endl << "main element id = " << mainId;
        cout << endl << "main element name = " << myControl.GetTag(mainId);

        // then we get the teacher's name ids...
        tempId = myControl.GetID(mainId, "teacher", 0);
        nameId = myControl.GetID(tempId, "name", 0);

        // and extract the teacher's firs and last names with
        // the ids obtained in the previous step...
        cout << endl << endl;
        cout << "Teacher: " << myControl.GetDataString(myControl.GetID(nameId, "first", 0));
        cout << " " << myControl.GetDataString(myControl.GetID(nameId, "last", 0));
        cout << endl;

        // after that we verify the number of students in the class...
        numDeAlunos = myControl.GetNumberOfElements(mainId, "student");

        cout << "Number of students in the class: " << numDeAlunos;
        cout << endl;
        cout << endl << "List of Students" << endl;

        // if there are any students...
        if(numDeAlunos > 0)
        {
            for(long i = 0; i < numDeAlunos; i++)
            {
                // we fetch an id for each student,...
                tempId = myControl.GetID(mainId, "student", i);
// then obtain the ids for each of their names...
                nameId = myControl.GetID(tempId, "name", 0);
                // and finally reach the name and last name data for each one...
                cout << endl << myControl.GetDataString(myControl.GetID(nameId, "first", 0));
                cout << " "<< myControl.GetDataString(myControl.GetID(nameId, "last", 0));
                // and also the grades for each subject...
                gradesId = myControl.GetID(tempId, "grades", 0);
                cout << endl << "\tGrades: " << endl;
                cout << "\t\tMath: " << myControl.GetDataString(myControl.GetID(gradesId, "math", 0));
                cout << endl << "\t\tArt: " << myControl.GetDataString(myControl.GetID(gradesId, "art", 0));
            }
        }

        cout << endl;
    }
}
```

Example 6 - Extracting data from the classroom file

Data Modification

The previous sections of this tutorial showed the most likely scenarios when working with MDSL files. You normally either read a file or write one, or maybe even do both operations but at different points in program execution. At times, however, you may wish to add a few elements to a file you just opened. For these situations you can use the same APIs used for file creation, except for `CreateMainElement()`, which can only be called once, on an empty data structure. In addition you may want to modify data in existing elements. For this you can call: `SetDataString(long elemId, string theStringData)`, `SetDataLong(long elemId, long theLongData)` or `SetDataDouble(long elemId, double theDoubleData)`, all of which return an error code on failure or `EPzeroERROR` when successful.

A word about efficiency

As you can see in the source code for the `MDSControl` class, the data insertion routines in the MDSL Library are very inefficient. This is because these functions were only added later for completeness. The main focus of the Library was always on file reading, which represents the most work when managing xml-type files. In order to speed up data retrieval when reading a file, `MDSControl` creates an internal address table which converts element id's directly to node addresses. This table is created at parsing time and normally does not add much overhead to loading the file. However, as this table is implemented as a dynamic array, when extra elements are added later, the table needs to be reallocated and copied for every new entry. This is still not a problem if you only add a few elements (say 100 or so). However, as the number of added elements grows, the time to copy that address table increases in the same proportion. So each addition becomes longer than the last one (last time I tested, my computer hang for a few seconds when trying to produce a class with 10,000 students with `AddNewElement`).

As we discussed in Creating MDSL Files, formatted text output can be easily achieved in C++ by using string objects and standard output streams. There is no real need for library calls when generating MDSL files from your app. So, as a rule, if your are planning to create long MDSL files, it is probably a good idea to write your own file writing routines. Now, if you only need to have a few elements in your file, you can use `MDSLControl` and forget about it.

Library Reference

MDSL Library (version 1.0)

Data Types

MDSL's main data type is the `MDSLControl` object. This object is responsible for opening and closing files, performing data checks on parsed documents and setting/retrieving data from the document's hierarchy tree. It relies on Standard C++ Class Libraries for file i/o and string handling.

MDSLControl

Description:

This object is used to control your application's access to the MDSL Library. An instance of the **MDSLControl** class should be declared once (and only once) before reading MDSL files. Every method in the Library can be called through this object, once you declare it. This object's constructors take care of initializing id counts, and other house keeping tasks.

Public Methods:

```
// Parsing
EP MDSLControl::OpenFile(string fileName, string structureListFile);

//Getting Info
long MDSLControl::GetNumberOfElements(long elemId);
long MDSLControl::GetNumberOfElements(long elemId, string elemTag);
long MDSLControl::GetMainID(void);
long MDSLControl::GetID(long fatherId, string elemTag, long index);
string MDSLControl::GetTag(long elemId);

// Getting Data
string MDSLControl::GetDataString(long elemId);
long MDSLControl::GetDataLong(long elemId);
double MDSLControl::GetDataDouble(long elemId);

// Setting Data
EP MDSLControl::AddNewElement(long fatherId, string elemTag, long index)
EP MDSLControl::SetDataString(long elemId, string dataString);
EP MDSLControl::SetDataLong(long elemId, long dataLong);
EP MDSLControl::SetDataDouble(long, double dataDouble);
EP MDSLControl::WriteToFile(string fileName);
```

Usage:

```
#include “MDSLControl.h”
MDSLControl myControl;
```

METHODS

This section describes `MDSLControl`'s public methods. For each method there is a description, prototype declarations, argument details and usage.

This reference only documents the public methods of the MDSL Library, the ones that can be used by external code for building data structures and handling files. The Library also implements a few routines that are for internal use only and therefore are not explained here. For those methods, please refer to source code.

The ***usage*** entries here are provided only to demonstrate specific syntax of these methods. They are not complete examples. Indeed, most of them need to be completed with other code in order to function in context. This is denoted by the `//...` marks before or after a group of statements. For a more complete example of how to use the MDSL Library, please refer to the TUTORIAL section on previous pages of this document.

READING FILES

OpenFile()

Prototype:

```
EP MDSLControl::OpenFile(String fileName, string structureList);
```

Description:

OpenFile opens an MDSL document and parses it to construct a data tree. During parsing, it looks for problems in the XML structure of the document and aborts with an error if any anomalies are found. If parsing concludes without errors, it then checks the tree against the structure list for application specific features of the data structure (element definitions, data types. etc.). Structure lists are optional. If a path for the structure list is provided in `structureList`, `openFile()` performs validation as described above. If, instead, an empty string is passed, `openFile()` bypasses validation.

* For more details on how to construct a structure list see **Structure Lists**, later in this Reference.

Arguments:

`fileName`	a string containing the full pathname for the MDS file to be opened. The data type for this may be either a C-style string (i.e.: a zero terminated array of characters) or a standard C++ Library string class object.
`structureList`	a string containing the full pathname for the structure list file. The data type for this may be either a C-style string (i.e.: a zero terminated array of characters) or a standard C++ Library string class object.

Usage:

```
EP error = EPzeroERROR;
MDSLControl myControl;
error = myControl.OpenFile("myFile.mds");
if(error != EPzeroERROR)
{
        // DO NOT USE THIS CONTROL IF OpenFile() RETURNS AN ERROR
}
```

Return Codes:

```
EPzeroERROR
EPinsuficientMemoryERROR
EPopenTagNotFoundERROR
EPinvalidOpenTagERROR
EPcloseTagNotFoundERROR
EPemptyTreeERROR
```

GetNumberOfElements( )

Prototype:

```
long MDSLControl::GetNumberOfElements(long fatherId);
```

Description:

Returns the number of child elements under a given element in the data tree. This is the total number of child elements, including all element types. For counting child elements of a given type see the overloaded version of this method in the next reference entry. This function should only be used after successfully parsing the file with `OpenFile()`.

Arguments:

`fatherId`	a long integer id number for the father element (i.e.: the element for which a child count is returned).

Usage:

```
long elemCount;
long fatherId
// ...
elemCount = myControl. GetNumberOfElements (fatherId);
```

Return value:

This method returns 0 if no child elements are found.

GetNumberOfElements( )

Prototype:

```
long MDSLControl::GetNumberOfElements(long fatherId, string tagName);
```

Description:

Returns the number of child elements of a specific type, embedded under a given element in the data tree. While counting, element tag names are compared to the string in `tagName`. Only exact matches (case sensitive) are counted. This function should only be used after successfully parsing the file with `OpenFile()`.

Arguments:

`fatherId`	a long integer id number for the father element (i.e.: the element for which a child count is returned)
`tagName`	a C++ string class object containing a tag name to be compared to each element's tag during counting. Tag names are case sensitive and only exact matches are counted.

Usage:

```
long elemCount;
long fatherId;
string theType;
// ...
elemCount = myControl.GetNumberOfElements (fatherId, theType);
```

Return:

This method returns 0 if no elements of type `fatherId` are found.

GetMainID( )

Prototype:

`long MDSLControl::GetMainID(void)`

Description:

Returns the id number for the top level element in the data tree -- the root element. This function should only be used after successfully parsing the file with `OpenFile()`.

Arguments:

`void`	This method takes no arguments.

Usage:

```
long topId;
// ...
topId = myControl.GetMainID();
```

Return:

This method returns the id number of the root element. This id should be retained for use with many `MDSLControl` calls.

GetID( )

Prototype:

```
long MDSLControl::GetID(long fatherId, string elemType, long index);
```

Description:

Returns the id number for the requested child element. If only one child element of the requested type is present under the element referred to by `fatherId`, it is accessed by index 0. When used in a loop, this method will return id numbers for each child element of the requested type, in the order they were loaded in the data tree during parsing. A call to `GetNumberOfElements()` will provide limits for index incrementing.

Arguments:

`fatherId`	a long integer id number for the father element. This is the element containing the requested element.
`elemType`	a string object containing a tag name to be compared to each element's tag. Tag names are case sensitive and only exact matches are considered.
`index`	a long integer number indicating the position of the requested child element in the father's children list.

Usage:

```
long id, childId;
long childCount;
//...
// First count the number of items of the requested type...
childCount = myControl.GetNumberOfElements(id, “UserRecord”);
// Then ask for each element id number...
for(long i = 0; i < childCount; i++)
{
       childId = myControl.GetID( id, “UserRecord”, i );
       // do something with each element...
```

Return value

This method returns the id number of the requested element. This id may be used with other `MDSControl` methods to retrieve information about an element. If there is no element of `elemType` at position `index` , the method returns `0` (obs.: `0` is not a valid id number)

GetTag( )

Prototype:

`string MDSLControl::GetTag(long elemId);`

Description:

Returns the name tag of the element referred to by `elemId`.

Arguments:

`elemId`	a long integer id number for the requested element. This number should be obtained by a previous call to `GetID()` or `GetMainID().`

Usage:

```
long theId;
string theTag;
// ...
theTag = myControl.GetTag(theId);
```

Return:

This method returns a C++ string class object. If no element is found with the id provided in `elemId`, this string object will be empty.

GetDataString( )

Prototype:

`string MDSLControl::GetDataString(long elemId);`

Description:

Returns the data content of the requested element. This is the string of characters that appears between an element's tags, in the MDS file.

Arguments:

`elemId`	a long integer id number for the requested element. This number should be obtained by a previous call to `GetID()` or `GetMainID()`.

Usage:

```
long theId;
string theData;
// ...
theData = myControl.GetDataString(theId);
```

Return:

This method returns a C++ string class object. If no element is found with the id provided in `elemId`, this string object will be empty.

GetDataLong( )

Prototype:

```
long MDSLControl::GetDataLong(long elemId);
```

Description:

Returns the data content of the requested element converted to a long integer number. Details of this conversion are described in the **Number Conversion** section of this document.

Arguments:

`elemId`	a long integer id number for the requested element. This number should be obtained by a previous call to `GetID()` or `GetMainID()`.

Usage:

```
long theId;
long theValue;
//...
theValue = myControl.GetDataLong(theId);
```

Return:

This method returns a long integer number. If no element is found with the id provided in `elemId`, this number will contain the value `0` (for more details see **Number Conversion**)

GetDataDouble( )

Prototype:

```
string MDSLControl::GetDataDouble(long elemId);
```

Description:

Returns the data content of the requested element converted to a double precision floating point number. Details of this conversion are described in the **Number Conversion** section of this document.

Arguments:

`elemId`	a long integer id number for the requested element. This number should be obtained by a previous call to `GetID()` or `GetMainID()`.

Usage:

```
long theId;
double theValue;
// ...
theValue = myControl.GetDataDouble(theId);
```

Return:

This method returns a double. If no element is found with the id provided in `elemId`, this number will contain the value `0`.

Note: Data setting and file writing routines are not mentioned here because they have not yet been implemented. Look for new versions of this document for a complete reference set.

CreateMainElement( )

Prototype:

`long MDSLControl::CreateMainElement(void);`

Description:

Creates the main element in the data tree and returns its id number. This method should only be called if the data tree is empty. If there are any previous elements inside the `MDSLControl`, `CreateMainElement()` returns -1.

Arguments:

`void`	this method takes no argument

Usage:

```
long mainId;
MDSLControl myControl;
EP ep = EPzeroERROR;
ep = myControl.OpenFile("myFile.mds","");
if(ep == EPzeroERROR)
{
        mainId = myControl.CreateMainElement(theId);
        if(mainId != -1)
        {
                // keep going...
        }
}
```

Return:

This method returns an the id number for the main element or -1 if main element could not be created.

AddNewElement( )

Prototype:

`EP MDSLControl::AddNewElement(long parentId, string tag, string data);`

Description:

Appends a new element to the children list of the element referred to by parentId. `tag` provides the tag name for the new element and `data` provides the element's data string. If the new item is to be a structural element, that is, an element with no data, an empty string (" ") should be passed as third argument.

Arguments:

`parentId`	the id number of the element whose children list is being appended.
`tag`	the tag name for the new element
`data`	the data string for the new element (if any)

Usage:

```
long parentId, childId;
// ...
childId = myControl.AddNewElement(parentId, "MyNewTag", "");
```

Return Codes:

`EPzeroERROR`

`EPinsuficientMemoryERROR`

`EPCouldntAddElementERROR`

SetDataString( )

Prototype:

```
EP MDSLControl::SetDataString(long elemId, string data);
```

Description:

Replaces the data string of the element referred to by `elemId` with the string contained in `data`.

Arguments:

`elemId`	the id number of the element whose data is to be modified
`data`	the new data for the element

Usage:

```
long elemId;
// ...
myControl.SetStringData(elemId, "New Data");
```

Return codes:

`EPzeroERROR`

`EPCouldntSetDataStringERROR`

SetDataLong( )

Prototype:

```
EP MDSLControl::SetDataString(long elemId, long data);
```

Description:

Replaces the data string of the element referred to by `elemId` with the value of `data` converted to a string.

Arguments:

`elemId`	the id number of the element whose data is to be modified
`data`	a long integer which will be converted to a C++ string object

Usage:

```
long elemId;
long num = 1234567;
// ...
myControl.SetDataDouble(elemId, num);
```

Return codes:

`EPzeroERROR`

`EPCouldntSetDataStringERROR`

SetDataDouble( )

Prototype:

```
EP MDSLControl::SetDataString(long elemId, double data);
```

Description:

Replaces the data string of the element referred to by `elemId` with the value of `data` converted to a string.

Arguments:

`elemId`	the id number of the element whose data is to be modified
`data`	a double precision number which will be converted to a C++ string object

Usage:

```
long elemId;
double num = 0.00123;
// ...
myControl.SetDataDouble(elemId, num);
```

Return codes:

`EPzeroERROR`

`EPCouldntSetDataStringERROR`

WriteToFile( )

Prototype:

```
EP MDSLControl::WriteToFile(string fileName);
```

Description:

Writes the entire data tree to a file. `fileName should be a string containing the full path to the target file.`

Arguments:

`elemId`	the id number of the element whose data is to be modified
`data`	a double precision number which will be converted to a C++ string object

Usage:

```
long elemId;
double num = 0.00123;
// ...
myControl.SetDataDouble(elemId, num);
```

Return codes:

`EPzeroERROR`
`EPCouldntSetDataStringERROR`

NUMBER CONVERSION

MDSControl converts data received from data elements to integer (long) and floating point (double) number formats. This section describes how this conversion is implemented and its consequences for data stored in the **MDSL** format.

Prior to converting strings to numbers, **MDSControl** checks to make sure there are no characters in the string other than digits, and, if necessary, a decimal point and/or a minor sign. Any other characters are considered illegal. For this reason, if an application needs to use scientific (e^10, etc.), or any other types of number notation, it will need to store/retrieve the numbers as strings and use its own number conversion routines.

Value checking in elements containing number strings is done at parse time (right after the data tree is loaded), and according to the rules set by the structure list (see next section). If a string is considered unsuitable for number conversion, when it should be a valid numeric field, the entire document will be considered unusable.

On the other hand, trying to convert any element data to numbers, while possible, will subject the data to the same restrictions. In this case, if the data contains any *non-numeric* characters, or if for any other reason, **MDSControl** cannot convert the data, the value `0` is returned.

While `0` is a perfectly valid and very common value, and therefore may not be the best fit for a default exception case, it was chosen as the path of least damage, short of increasing the complexity of the library's programming interface. Thus, encountering uncommonly frequent successions of zeroes can serve as cue to errors, and should be considered a sign that data may be textually corrupt.

Syntax of a Structure List

An **MDSL Structure List** is a list of element definition lines that serve as a base for validating an **MDSL** document of a given application. Typically an application will have an associated structure list for each type of document it supports. If for any reason the document specification changes, the corresponding structure list must be updated accordingly, so the parser can do its job. In other words, once the desired structure of the document is defined, a single structure list file should be written and made available to the parser.

The **Structure List** file must be present within the application's directory structure or a full path must be provided, so that it can be loaded at parse time. The following rules explain how a **Structure List** must be formatted:

Formatting Rules:

1. Each line in an **MDSL** structure list defines one type of element. This definition must be unique within an **MDSL** application. The definition lines contain space separated fields of number codes and character strings, as explained bellow.

2. The very first element definition must be the main (root) element in the data tree.

3. Each line starts with an index number, which is used to reference the occurrence of the element within other definition lines. The main element must have an index value of `0` (zero).

4. Standard C++ comments ("//") are allowed, in order to organize/document the structure list. Commented characters are ignored up to the next return ("\n").

5. After the index number there are four other required fields: *Tag Name, Mode, Count and Text Type*. Other fields may also occur depending on the value of *Mode*. Following is a detailed description of a definition line and an example of structure list based on the classroom application illustrated in the TUTORIAL section of this manual.

Syntax:

```
[INDEX] [TAG_NAME] [MODE] [COUNT] [TEXT_TYPE] [NUM_OF_TYPES] [CHILD_TYPES]
```

Field descriptions:

`Index`	A positive integer indicating the order in which each element definition line is loaded into memory. The main element always has an index value of `0`. The other element definition lines may be loaded in any order.
`Tag_Name`	The name of the element as found in the element's name tag in the **MDSL** document, but without the tag marks ('<', '>')
`Mode`	A positive integer code indicating what kind of content should be expected inside an element. Data elements (elements containing only text data) are represented by the value `0`. Elements which may contain other elements are identified by the value `1` in this field.
`Count`	The number of elements of this type allowed in the same level of the data tree (i.e.: the same child list). This field can be set to either 1 (if only one element of this type is permitted), or `0` (if multiple elements of this type are allowed).
`Text_Type`	A positive integer code indicating what kind of text should be expected inside an element of this type. if set to `0`, any text string (UTF 8 encoding) is allowed. If set to `1`, any elements of this type will have its data strings tested for numeric format. If this test fails, the entire document is deemed corrupt. Obs.: This field is only used if `Mode` is set to `0.` In non-data elements, this should be set to `0`.
`Num_Of_Types`	If this element contains other elements (`Mode`=`1`), `Num_Of_Types` shows the number of element types permitted in this element's child list. This number is just an item count for the field bellow. It tells **MDSControl** how many more parameters it should look for, in the remainder of the definition line. If Mode is set to 0, this number should not be present in the definition line.
`Child_Types`	A sequence of space-separated index numbers, referencing each element definition that can be found inside this element's child list (ex.: `[3 4 5 21 14]`). **MDSControl** looks in the rest of the structure list to identify the element definitions by their index. If these numbers are incorrect, the entire structure will be invalid and every document compared to this list will fail validation. If *Mode* is set to `0`, this sequence is not present.

Example:

```
// Main Element
0 Class 1 1 0 2 1 2

// Children of Class
1 Teacher 1 1 0 1 3
2 Student 1 0 0 2 3 6

// Generic elements
3 Name 1 1 0 2 4 5
4 First 0 1 0
5 Last 0 1 0
6 Grades 1 1 0 2 7 8
7 Math 0 1 1
8 Art 0 1 1
```

The example above illustrates the structure list for the '`class`' application, shown in **Figure 1** of the TUTORIAL. The list shows that, except for the `Student` element, every element can only appear once in its containing child list. Also, data elements like '`First`' or '`Math`' have shorter definition lines. The reason for this is that, having been defined as data elements (`Mode=0`), they don't need an index sequence for contained elements.

Revision History

19/02/2016	- Total Revision: introductory text and specs completely rewritten; added reference sections for data writing APIs; fixed names and reviewed all reference documentation; fixed errors in Structure List example.
31/03/2011	- Revision (fixed typos; corrected info about xml tag - page 4)
02/01/2011	- Revision (fixed minor typos, replaced illustrations)
07/10/2003	- First Draft