Reading & Writing Sensor Independent XML (0.1)

Table of Contents

About

The Sensor Independent XML library (six), is a cross-platform C++ API for reading and writing NGA's complex and derived sensor independent radar formats. Additionally it is the official reference implementation library for the Sensor Independent Derived Data (SIDD) format. The library is intended to be easy to use and integrate, without requiring extensive knowledge of XML or the underlying file formats. File format implementation concerns are handled separately from the data model, away from the end user. To create or read valid sensor independent formats using this library, the application developer is only concerned with populating the data model correctly. Reading and writing to NITF, GeoTIFF and XML are handled by the library internally. This also allows the library to be extended in the future to support new container formats or specification enhancements.

To facilitate extension and keep the code and coding mistakes to a minimum, the library uses external open source libraries to handle file format interactions. To minimize errors in application development, the design incorporates some common C++ design patterns However, to make the system easy to use for researchers, the main components of the data model are implemented purely as C-style structures.

The six library is available as open-source software under the Lesser GNU Public License (LGPL). This license is commonly used in the open-source community, and allows applications that are not open source to make use of the library without penalty. As with other open source projects, the library is available as-is, with no warranty.

Downloading the Library

The six library is currently distributed with limited-access from our subversion repository, located at https://wush.net/svn/gdais-oss[1]. The repository internally links to the GD-AIS CODA-OSS and NITRO projects on Sourceforge.net, using svn:externals properties. Several CODA-OSS modules are used as a portable runtime, and NITRO is used to read and write NITF files. When you checkout the subversion repository, subversion will automatically anonymously check out the CODA-OSS dependencies for you and put them in the proper place in your sandbox.

To checkout the repository, the user must have a login for that site. The library may also be packaged as a compressed source bundle, or in some cases as a binary. In order to checkout the code from subversion, you must have an svn client installed. On unix, the client is named 'svn.' To check out the repository from the top, execute the following command.

```
$ svn co https://wush.net/svn/gdais-oss/
```

You will be prompted for your username and password. If your username on the server is different from your system, hit return when prompted for a password and you will be prompted for an alternate username and password.

Building and Installing the Library

The library requires a C++ compiler. Most modern C++ compilers should work. The library does make use of templates, exceptions, and namespaces, so old or non-standards compliant compilers should be used with caution. The library can be built using the traditional unix configure/make system on Unix/Linux. On Windows, Visual Studio can be used to build a library. Additionally, we provide support for a cross-platform build tool called 'Waf,' which allows the same build scripts to be executed on Unix and Windows.

Building for Unix/Linux using configure/make

Most Unix developers will be familiar with configure/make. Our system does not vary too much from the ordinary:

```
$ cd trunk/modules/c++
$ ./configure --help
$ ./configure [options] [--prefix /path/to/install]
$ make
$ [make test]
$ make install
```

The XML handler library used by the library (called xml.lite) is a lightweight implementation of a simple Document Object Model (DOM) using one of several freely available popular open source XML libraries. At build configuration time, the developer can tell xml.lite to use expat, Apache Xerces-C++, or another library (check at configure time to see the latest options).

By default, six will use the open source 'expat' library to do XML parsing. Expat is a fast, low-overhead C XML parser with a capability set that works extremely well with the code base. If the build system finds expat on the target system, it will attempt to use the installed version. If it does not, it will attempt to build it from scratch from an internally bundled driver the first time the library is built. If expat already is installed on your system, you can use the --with-xml-home option to set the proper include and lib paths in the generated makefiles. You may optionally elect to use the Apache Xerces C parser instead. To switch the xml library to Apache's Xerces C parser, use the --enable-xml-layer=xerces in conjunction with the --with-xml-home path, identifying the location of Xerces, if it is not in your sytem path.

The NITRO library is GD-AIS software that aims to be a complete implementation of the NITF MIL-STD2500C. It is cross-platform, widely-used, and under active development and maintenance. The NITRO C library is bundled as a driver, and built on first use, if NITRO is not already detected on the target system when configure runs.

The SIDD file format optionally does support GeoTIFF. It currently uses the tiff module in CODA-OSS, which is a minimalist implementation of the format. It may be necessary in the future to switch this to use libtiff, but for now, the library is sufficient. SICD application developers will probably not be interested in the tiff support. Even though configure/make will currently build and link tiff for the test cases, it is possible to disable tiff support explicitly using by defining '--with-defs=SIX_TIFF_DISABLED' when building. If this is done during configuration, the application will no longer need to link against tiff-c++.

To build the test cases, run the 'make test' command. If you build the tests the binaries will be installed under the 'bin' directory.

> **Note:** The current configure/make system does not attempt to follow the typical OS conventions regarding library installs. The main difference is that it inserts a target directory under the libs and bin area containing the OS name as identified by config.guess. For this reason, it is recommended that you use waf to build if you intend to install to a system directory

Building for Windows using Visual Studio

<Fill this in as time allows>

Building on a System with Waf

<Fill this in as time allows>

Library Design Idioms, Rules and Conventions

While every attempt was made to keep the library minimalist, supporting both existing sensor formats in one library requires a fair amount of code. To make it easy to maintain and use with the library, certain idiomatic coding conventions are used. This section attempts to document these conventions and their rationale.

Naming and Type Conventions

Classes and structs follow Java style naming conventions. There first letter is always upper case, and their methods and fields always begin with a lower case letter. Camel-casing is always used. Fields in a class are primarily protected or private, and are prefixed by an 'm', denoting that they are member fields. Struct fields are always scoped as public and the fields are not prefixed as they are in classes. Most types used in reading and writing have overloaded methods to allow them to be printed as a string, or read in from strings. Basic primitives and enumerations are used where possible to minimize the code base and to provide a consistent API. Where these already exist in the CODA-OSS runtime, they are reused. Types that are taken from other CODA-OSS modules are often typedef'ed within six, to allow us to change their implementations in the future as necessary.

Data Model Conventions

The library tries to implement the idea of Separation of Concerns (SoC), which says that an object should focus on only one thing and do it well. This minimizes bugs and creates weak-coupling and strong cohesion. With respect to the blocks described by SICD and SIDD, we keep the object representation separate from the actions that transform it. In fact, to make things as simple as possible, C-style structs are used to represent all of the components that are present in the SICD and SIDD formats. This, theoretically, also allows the same capabilities to be extended for other packaging mechanisms, but it also just makes it easier for an application developer to write code.

Most modern XML binding frameworks generate data models that are tightly coupled with how they are represented in XML. Here we have made an explicit effort to separate the two. Certainly the XML reader and writer need to know what a ComplexData object looks like, but in most cases, that should be hidden from the application itself. Additionally, most XML bindings create objects that are based on the names of the XMLSchema-defined types. Here again, we try to keep the application developer programming to something that looks like the XML instance, so that he or she doesnt have to thumb through the schema every time looking for the type names. So for example, if a type in the XMLSchema is identified as 'exploitationfeaturescollectionphenomenologytype,' but its only existing usage in the output is as a 'Phenomenology' instance, a typical binding library would generate a type with the name given by the former, whereas, in six, it the struct will have the instance name (Phenomenology).

Resource Initialization, Cloning and Destruction

Data model objects are initialized as soon as possible. This usually means that if there are choices contained in the building of a complex object, an enumeration will be used to tell it how to initialize. We also require that an object in the data model with a required sub-field allocate its field as soon as it is created. Allocations for single objects are done using 'new.' C++ STL vectors are used to store unbounded objects. Any objects contained in a vector that are pointers are also allocated with the 'new' keyword. If a required sub-field has one or more elements, we initialize one element in the subvector. Objects with optional types are not initialized in the constructor. They are either set to a special undefined state (for non-pointers) or they are set to NULL if they

are pointers. Pointers are used for objects that have many configuration details, or where an undefined state cannot be determined without using NULL. There is only one way to tell if a pointer object is initialized in the library. A NULL-comparison is used. Therefore, all constructors must NULL-set any non-required pointers. An object in the data model owns its sub-fields. It is expected to delete them. If an application developer wishes to own a copy of some object within a data model, it must create it from scratch, or clone() it from an object in the model.

All objects in the data model must provide a clone method. The clone method requires them to make a deep copy of themself (this is sometimes called a 'prototype' design pattern). Prior to cloning, any sub-field pointers are compared against NULL first. If a sub-field object is NULL, the cloned object will also contain a NULL sub-field.

Since an object in the data model owns its own children, destructors are required to test each pointer type against NULL, prior to deletion. Non-NULL objects are freed using the 'delete' keyword.

Optional elements in the model should be allocated by the application using the 'new' keyword. Once its pointer is assigned in the sub-field of a data model object, that object owns it and will be responsible for deleting it. The caller should not delete this optional element, unless it first sets the parent's pointer field for that element back to NULL. The provided ComplexDataBuilder and DerivedDataBuilder objects will allow the application to create any optional top-level blocks in a Data object.

Actor Conventions and Patterns

Things that are not in the data model are not modeled as structs. They are C++ classes. For basic use cases, a user only encounters these objects when reading or writing a file. Actors make use of sub-classing as much as possible, to make it easier for applications to be flexible. For example, the ReadControl interface defines the contract for all SICD/SIDD read capabilities. To read a NITF, a NITFReadControl sub-class would be created. The same is true for WriteControl's contract. Actors can be new-allocated, or created on the stack if the implementation is known upfront. The example below shows good encapsulation of WriteControl creation, where it is separate from usage:

```
// Trivial example of factory-type set-up for a WriteControl
void createWriter(std::string siddFile)
{
    six::WriteControl* writer = NULL;

    // This example uses NITRO, but we could just check the extension
    if ( nitf::Reader::getNITFVersion(siddFile) == NITF_VER_UNKNOWN )
    {
        writer = new six::NITFWriteControl();
    }
    else
    {
        // Assume for this example it would be this otherwise
        writer = new six::GeoTIFFWriteControl();
    }
    return writer;
}
```

The XMLControl interface provides another contract, one that is responsible for serialization to and from a DOM, which is a standard for XML manipulation. XMLControl may be used independently from the ReadControl and WriteControl, since its job is not to write a file, but to change representations. Derived implementations exist for DerivedData and ComplexData, called DerivedXMLControl and ComplexXMLControl, respectively. These can be instantiated directly, or using the XMLControlFactory if the Data sub-class is not known by the caller.

```
// ComplexData* data

// Generically, from a Data*
six::XMLControl *genericControl =
    six::XMLControlFactory::newXMLControl(data->getDataClass());



// Concretely if we know what Data type we have
xml::lite::Document* dom = six::ComplexXMLControl().toXML(data);

// Or backwards
ComplexData* complexData = (ComplexData*) genericControl->fromXML(dom);

// Dump DOM to standard out
io::StandardOutStream;
dom->getRootElement()->prettyPrint(stream);

// Be a good citizen
delete dom;
delete genericControl;
```

Utilities exist to convert DOM data into a byte array or std::string as well:

```
char* charArray = six::toXMLCharArray(data);
std::string str = six::toXMLString(data);
...
// Be a good citizen
delete [] charArray;
```

The DerivedData object can be rigorously created using pointer assignments in the model. Alternately, there is an implementation of the Builder design pattern built into the library which can help. The builders for 0.1 do not allow the user to build every single component in the data model. For details, consult the doxygen:

```
six::ComplexDataBuilder cdb;

// Operators can be chained.  Each function updates the model
cdb.addImageCreation().addImageData(six::RE32F_IM32F);

// Get a ownership over the builder's data
six::ComplexData* cd = cdb.steal();
```

Using six to Read/Write Sensor Independent Derived Data (SIDD)

SIDD currently supports two file formats, NITF 2.1 and GeoTIFF, typically referred to as "containers" in document jargon. A SIDD file stored in either of these formats contains an XML meta-data section described by the SIDD XML Schema. It is possible to use the six library to read and write meta-data and imagery from either type of file, as well as to load or store its meta-data to or from a raw XML document.

Reading

Reading is multi-pass. The library reads meta-data using the load() function. To read image data, the interleaved() function should be used. The name alludes to the fact that multi-band data is read into a buffer in band interleaved order, as opposed to providing the developer with a separate buffer for each band. The latter method may be supported in the future using another API call.

The SICD file format only allows one SICD XML and image per file format container (see the SICD File Format Design Document for details). The SIDD format allows

multiple SICD and SIDD XML data and multiple derived images within a container (see the SIDD File Format Design Document for details). The six ReadControl's contract is modeled in a way that it does not care whether the data is SICD or SIDD. Currently, since SICD has only one defined official container format, NITF, the NITFReadControl is the only derived reader that can be used for SICD data. The NITFReadControl and GeoTIFFReadControl may be used to read SIDD data. The interface does not change, irrespective of container format or sensor data format. Here is an example of reading SICD meta-data from a read control:

```
// Create the reader
six::NITFReadControl reader;

// Load the image, bind it to a container
reader.load("/path/to/sicd.nitf");

// Get the container
six::Container* c = reader.getContainer();

// Get the SICD data model
six::ComplexData* sicd = (six::ComplexData*)c->getData(0);

// Read the number of rows using Data* base virtual method
std::cout << sicd->getNumRows() << std::endl;

//-------------------------------------------------------------
// Read the number of rows using ComplexData* structure
// Note this corresponds roughly to the XPath statement:
// /SICD/ImageData/NumRows
//-------------------------------------------------------------
std::cout << sicd->imageData->numRows << std::endl;
```

The same method can be used to read in SIDD DerivedData. The SIDD specification allows the file format container to include SICD XML about the collection, and one or more SIDD sections as well. It is up to a "profile" to determine how this data is ordered. For the purposes of our example, we will say that the profile for this particular data specifies that there will be one SIDD XML followed by its parent products' SICD XML. Since this is a SIDD, it could be a NITF or a GeoTIFF, but lets imagine that the profile only allows a NITF for this particular type of product:

```
//-------------------------------------------------------------
// This code does not rigorously check that the product
// actually matches its profile, it assumes that it is valid
//-------------------------------------------------------------

// Create the reader
six::NITFReadControl reader;

// Load the image, bind it to a container
reader.load("/path/to/sidd.nitf");

// Get the container
six::Container* c = reader.getContainer();

// Get the SIDD data model (first data block)
six::DerivedData* sidd = (six::DerivedData*)c->getData(0);

// Read the number of rows using Data* base virtual method
std::cout << sidd->getNumRows() << std::endl;

//-------------------------------------------------------------
// Read the number of rows using DerivedData* structure
// Note this corresponds roughly to XPath statement:
// /SIDD/Measurement/PixelFootprint/Row
//-------------------------------------------------------------
```

```
std::cout << sicd->measurement->pixelFootprint.row << std::endl;

// Get the SICD data model (second data block)
six::ComplexData* sicd = (six::ComplexData*)c->getData(1);
...
```

Due to the details of the container formats, some difference exist for derived ReadControls. In the case of NITF, a single image segment size is limited to at most, around 10GB of data. The TIFF file format itself is limited to 4GB in its entirety. To circumvent the NITF issue, both SICD and SIDD file format documentations define an algorithm for properly segmenting one continuous image into multiple segments within the NITF. Since neither format uses TREs, this is relatively straightforward as far as meta-data is concerned, but it does require some book-keeping, and some updates to the NITF CCS. A goal of the six library is to hide the details of the file format container as much as possible. To that end, the library hides segmentation rules from the application developer, and instead, presents the developer with access to one virtual SICD image, irrespective of how many segments exist in the NITF. With SIDD images, each image may end up segmented (this is very unlikely to occur in real-life scenarios), and each product is presented as a virtual SIDD image, similar to the SICD usage. Therefore, the application developer will have no need to access the NITF image segment data, since the XML contains the meta-data for the full pixel array.

SICD pixel arrays contain a band for real and a band for imaginary data. The data is represented with each pixel made up of one real component and one imaginary. In NITF jargon, this is called 'band-interleaved by pixel.' SIDD 24-bit RGB pixel arrays are interleaved the same way, where each pixel is organized one after another as a byte of red, a byte of green, and a byte of blue, followed by the next pixel. SIDD monchrome data and RGB 8-bit LUT data is presented as a single band. In the latter case, the 8-bit value is an index into the value:

```
// Print a LUT for a given indexPixel
six::LUT* lut = sidd->display->remapInformation->remapLUT;
char* pixel = (*lut)[indexPixel];
short r = (short)pixel[0];
short g = (short)pixel[1];
short b = (short)pixel[2];
std::cout << "R,G,B: " << r << ',' << g << ','<< b << std"::endl;
```

Reading interleaved data using a ReadControl is done using a Region object, which specifies a window to be read. The caller can allocate a buffer for reading, and reuse it at a later point. Otherwise, the library will create a buffer of the appropriate size, and give it back to the user. If the caller allocates its own memory, it should be careful to ensure that it has allocated the right amount prior to calling the interleaved() function, in order to avoid memory corruption. The requested window size is established in the Region. If the numRows and numCols are not set, or are set to -1, the ReadControl will read from the start points up to the end of the image into the buffer, and then sets the actual numRows and numCols in the region prior to returning:

```
// Read one row at a time from a SICD container, and write to a file
Region region;
region.setNumRows(1);

// Get the first data object
Data* sicdData = container->getData(0);

// Allocate a one line buffer to read complex data
unsigned long nbpp = sicdData->getNumBytesPerPixel();
unsigned long height = sicdData->getNumRows();
unsigned long width = sicdData->getNumCols();
unsigned long nbpr = nbpp * width;
workBuffer = new UByte[nbpr];
```

```
// Set it in the region
region.setBuffer(workBuffer);

// Set up an output file
io::FileOutputStream outputStream(outputFile);

// For each line in the image
for (unsigned int i = 0; i < height; i++)
{
    // Update the start row, other than that
    // no need to modify region, we are okay
    region.setStartRow(i);
    UByte* line = reader->interleaved(region, 0);
    outputStream.write((const sys::byte*) line, nbpr);
}
// Must delete our work buffer
delete [] workBuffer;
// Delete or ReadControl
delete reader;
// Close our output stream
outputStream.close();
```

It might be convenient in some cases to allow the library to allocate enough memory, especially if we are going to read up to the number of rows and columns all at once:

```
// Read a chip out of the image
region.setStartRow(startRow);
region.setEndRow(endRow);
region.setStartCol(startCol);
region.setEndCol(endCol);

std::complex<float>* chip =
    (std::complex<float>*)reader->interleaved(region, 0);

...

// Still have to do this at the end
delete [] region.getBuffer();
```

Writing

To write, the application developer will use the WriteControl contract, using the appropriate sub-class. It is important that the contents of the Container be filled and organized in the order that it is to be written in the file. There should be only one Data inside a SICD Container, and there will be only one image source passed into the save() function. For SIDD Container's, all meta-data will be written out, but the number of SIDD Data objects will correspond one-to-one with the number of image sources provided to the save() call. There are two types of image sources that the WriteControl accepts. The first is a byte buffer. This makes it easy to write an in-memory pixel array out to a SICD or SIDD. The second method is via the io::InputStream interface. In short, any class that derives InputStream can be used as an image source for writing, including io::FileInputStream, io::ByteStream, and any source that an application developer can adapt to a stream. Creating a new InputStream sub-class is trivial, since only two functions must be overriden (one telling how many bytes are in the stream, and one to read N bytes from the stream. The example below shows how to write some unidentified stream into a SIDD image pixel array:

```
//-------------------------------------------------------------
// Assume the Container is properly set up
```

```
// and that we have two images in this SIDD product
// That means the container has two SIDD instance already
// Set up (perhaps some ComplexData as well
//-------------------------------------------------------------

// See create example above for pseudo-impl
WriteControl* writer = createWriter(file);

// Set up the container.
writer.initialize(container);

// We have a source for each image
std::vector<io::InputStream*> sources;

//-------------------------------------------------------------
// Some function to create a derived impl of InputStream
// Lets say this source function needs to know pixel array
// information contained in the DerivedData section
//-------------------------------------------------------------
io::InputStream* is1 =
    createInputSourceForProduct1(params, container->getData(0));

// Same type of function, but needs MD for SIDD data 1, not 0
io::InputStream* is2 =
    createInputSourceForProduct1(params, container->getData(1));

sources.push_back( is1 );
sources.push_back( is2 );

// Save the file
writer->save(sources, outputName);

// Delete the sources
for (unsigned int i = 0; i < sources.size(); ++i)
    delete sources[i];
```

Getting Help and Reporting Bugs

This library is provided as-is. This document is meant to help developers, but when in doubt, the online Doxygen generate API documentation or, if necesary, source code should be consulted. If you fix a bug or make enhancements to this freely provided source code, the LGPL requires that you submit your changes back to the copyright holders.

Software bugs should be reported using the Trac issue tracker (currenlty https://wush.net/trac/gdais-oss/) available at the current project hosting location. To post an issue, you must currently have a valid login. If you are using a guest account, please provide us with a name and email or phone contact information in the issue. When reporting a bug, try to be as specific as possible about the problem you are having. Please also provide an example of what you are doing. If your code is on a different network, or non-disclosable, please attempt to make a short, concise example that would replicate the problem you are seeing. We do not currently have an email group for help or discussion. If you email the POC for the SIDD development effort, we will make every attempt to your request properly. If you arent sure, you can post an issue on the Trac for the time being. Please check the FAQ page before posting.

Notes

1. https://wush.net/svn/gdais-oss/
2. https://wush.net/trac/gdais-oss/