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 source 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 a user does a checkout from the subversion repository, subversion will automatically anonymously check out the *CODA-OSS* dependencies for and put them in the proper place in the 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, a subversion client must be installed. On Unix, the *svn* command does the job. 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 *configure/make* (*autotools*) 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 as a driver. 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 *expat* library to do XML parsing. *Expat* is a fast, low-overhead C XML parser 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 the *Xerces* 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 the 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 internally and automatically built on first use, if it is not already detected on the target system when *configure* is run.

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.

> **Note:** It is likely that the library will support the standard *libtiff* library as a driver in the future.

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* during a *configure*. If this is done during configuration, the application will no longer require the *tiff* driver.

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 *lib* 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 and Unix with *Waf*

The *waf* build system is cross-platform, since it is written in Python. To run *waf*, you need an installation of Python with *bz2* support built-in. This is standard for most Python installs, but if you do not have it, *waf* will not unpack itself correctly. The *waf* build system for *six* is slightly different than the *configure/make* system, but it presents the user with uniform behavior across all systems. The biggest difference is that waf builds on Windows.

In order to build on Windows, you will need a *Visual Studio Compiler*. *Microsoft Visual Studio C++ Express Edition* is freely available, and is what the developers use to test *six* on Windows. To build within a DOS shell, you may need to run the *vcvars32.bat*.

> **Note:** The version of *waf* that *six* currently bundles has problems with whitespace in path names, so in this example, we will install it to a path with no whitespace. It is presumed that future versions of *waf* will or have aready resolved this issue:

To build, first get a complete checkout. Tools such as TortoiseSVN make it easy to check out code from the repository on Windows. Once you have a sandbox, and can run Python on your Windows machine, you are ready to build from a DOS shell. The first step is to run *waf configure*. The *waf* program is a Python script, and should either be run as an argument to Python, or you should make sure that the *python* executable is in your path.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd C:\gdais-oss\gdais-oss\trunk\modules

C:\gdais-oss\gdais-oss\trunk\modules> python waf configure --prefix=C:\gdais-oss\instal
Checking for platform                    : win32
Checking for program CL                  : ok C:\Program Files\Microsoft Visual
Studio 9.0\VC\BIN\CL.exe
Checking for program CL                  : ok C:\Program Files\Microsoft Visual
Studio 9.0\VC\BIN\CL.exe
Checking for program LINK                : ok C:\Program Files\Microsoft Visual
Studio 9.0\VC\BIN\LINK.exe
Checking for program LIB                 : ok C:\Program Files\Microsoft Visual
Studio 9.0\VC\BIN\LIB.exe
Checking for program MT                  : ok C:\Program Files\Microsoft SDKs\Wi
ndows\v6.0A\bin\MT.exe
Checking for program RC                  : ok C:\Program Files\Microsoft SDKs\Wi
ndows\v6.0A\bin\RC.exe
Checking for msvc                        : ok
Checking for msvc                        : ok
Checking for header inttypes.h           : not found
Checking for header unistd.h             : not found
Checking for header getopt.h             : not found
Checking for header malloc.h             : ok
Checking for header sys/time.h           : not found
Checking for header dlfcn.h              : not found
Checking for header fcntl.h              : ok
Checking for header check.h              : not found
Checking for header memory.h             : ok
Checking for header string.h             : ok
Checking for header strings.h            : not found
Checking for header stdbool.h            : not found
Checking for header stdlib.h             : ok
Checking for header stddef.h             : ok
Checking for function mmap               : not found
Checking for function memmove            : ok
Checking for function strerror           : ok
Checking for function bcopy              : not found
```

```
Checking for type size_t                 : ok
Checking for const keyword               : ok
Checking for unsigned short              : ok
Checking for unsigned char               : ok
Checking for library m                   : not found
Checking for library sqrt                : not found
Checking for function gettimeofday       : not found
Checking for function BSDgettimeofday    : not found
Checking for function gethrtime          : not found
Checking for function getpagesize        : not found
Checking for function getopt             : not found
Checking for function getopt_long        : not found
Checking for function isnan              : not found
Checking for type hrtime_t               : not found
Checking system type sizes               : ok
Checking for sizeof long long            : 8
Checking for sizeof long                 : 4
Checking for bigendian                   : False
Checking for sizeof double               : 8
Checking for sizeof short                : 2
Checking for sizeof int                  : 4
Checking for sizeof float                : 4
Checking for library expat               : not found
Building local lib                       : expat
'configure' finished successfully (13.250s)

C:\gdais-oss\gdais-oss\trunk\modules>
```

The second step is to call the *waf build* or just *waf* (they do the same thing).

```
C:\gdais-oss\gdais-oss\trunk\modules> python waf
...
...
'build' finished successfully (4m3.592s)
```

Now we have to install the library. The *waf* scripts do not insert the build target name under the *bin* or *lib* directory on an install, unlike the *configure/make* system. If you are installing this library into a system path, you will most likely prefer the install done by the *waf* scripting. Users attempting to install many different targets into the same path using only one set of includes will likely prefer the *configure/make* method. Prior to install, the waf scripting does place targets under their corresponding platform name in the *target* directory.

```
C:\gdais-oss\gdais-oss\trunk\modules> python waf install
```

Unix builds look more or less the same. Most of the time on Unix, it is not necessary to explicitly invoke *python* at the beginning of the command (although it is possible). Here are the commands run from a Linux box:

```
user.machine gdais-oss/trunk/modules $ ./waf configure --enable64bit --prefix install
...
'configure' finished successfully (3.438s)
Waf: Leaving directory `/home/user/gdais-oss/trunk/modules/target'
user.machine gdais-oss/trunk/modules $ ./waf
...
'build' finished successfully (32.147s)
user.machine gdais-oss/trunk/modules $ ./waf install
...
'install' finished successfully (1.650s)
```

Note that on some AMD chipsets, you may have to specify *--enable-64bit* explicitly or you may get a link-error between some of the compiled components.

Building for Windows using Visual Studio

It should be possible to create a project solution using the Visual Studio graphical interface. The recommended way is to compile each module separately as a library, or together in one large library, separate from the application code. A solution file is not currently provided for the library, although the *waf* 'wscripts' do use the Visual Studio Compiler to build on Windows, and thus contain all of the flags and preprocessor definitions that are necessary to build using the Visual Studio compiler. Any user wishing to build on Windows without using waf should consult the build scripts for more detail.

Library Design Idioms, Rules and Conventions

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

Naming and Type Conventions

Classes and structures follow Java style naming conventions. The first letter is always upper case, and the class 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. Structure fields are always public scope and the fields are not prefixed as they are in classes. Most types in the library have overloaded methods to allow them to be printed as a string, or read in from strings, using the *str::toString()* and *str::toType()* methods . 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) -- the concept that an object should focus on only one thing and do it well. This minimizes bugs and creates loose-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. The former is called the "data model" and the latter are called "actors" or "controls." To make things as simple as possible, C-style structures are used to represent all of the components in the data model for SICD and SIDD formats. This, theoretically, allows the actors to be extended for other packaging mechanisms, but it also just makes it easier for an application developer to write code.

Most modern XML code-generation binding frameworks generate data models that are tightly coupled with their XML representations. Here we have made an explicit effort to separate the model from how it is serialized. 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, and from the data model itself.

Additionally, most XML binding frameworks create objects that are based on the names of the XMLSchema-defined types. This can be helpful when dealing with schemas that have duplicate element names, and lots of dependencies. The *six* library tries to keep the data model closer to something that looks more like the XML instance, so that the application developer does not have to thumb through the schema every time looking for the type names.

For example, one type in the SIDD schema is identified as *exploitationfeaturescollectionphenomenologytype*, but its single existing usage within SIDD is as a *Phenomenology* element in the XML instance. A typical binding library would generate an *exploitationfeaturescollectionphenomenologytype* structure, and the parent would have a pointer to a *phenomenology* property, but that would mean that any function that acts on the type would need to reference the type name, not the instance name. We felt that it was less confusing for the application to pass a *Phenomenology* structure instead, and keep the property in the parent as *phenomenology* as well, mirroring the name as it appears in the actual XML data.

> **Note:** There are places in the data model where the property or class name deviates slightly from the name as it appears in the XML instance. These differences usually arise from the desire to keep a consistent API. Each occurence is documented individually in the *doxygen* comments along with the actual SICD or SIDD element name.

Resource Initialization, Cloning and Destruction

The data model root object is called *Data*, and its sub-class implementations for SICD (*ComplexData*) and SIDD (*DerivedData*). The *Data* class provides a minimal contract to its sub-classes, including some utility methods to get information that is required for both derived classes, but may live in different places in each objects XML representation. Each derived *Data* class has pointers to its XML blocks (top level elements). Every object underneath a derived *Data* implementation is part of the data model.

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 passed to the constructor to tell the object how to initialize. An object in the data model with a required sub-field is required to allocate the sub-structure 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, the library will initialize one element in the sub-vector.

Objects with optional types are not initialized in the constructor. They are either set to a special undefined state or 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. A NULL-comparison is the only way to determine if a pointer object is initialized in the library. Therefore, all constructors must NULL-initialize 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. Optional and unbounded sub-fields are represented by an empty STL vector. It is up to the application to provide optional elements by inserting the into the vector as needed.

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.

All objects in the data model must provide a *clone()* method. The *clone()* method contract requires that the object make a deep copy of itself (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.

Actor Conventions and Patterns

Actors (also referred to in the library as controls) are object that perform an action on a data model. They are not part of the model and are implemented as C++ classes. Most often, the developer encounters these objects when reading or writing a file. Actors often define a contractual interface, and sub-classes provide the implementation wherever possible. The interface/implementation convention is more flexible, and makes it easier to extend the library as necessary. 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. To read a GeoTIFF, the *GeoTIFFReadControl* sub-class is used. The same is true for *WriteControl*'s contract. Actors can be new-allocated with a factory method, or created on the stack if the implementation is known upfront. The example below shows good encapsulation of *WriteControl* creation, where the instantiation 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 a contract for serialization of a data model to and from a DOM, -- a standard structure 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* subclass 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 stream;
dom->getRootElement()->prettyPrint(stream);
```

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

Since the library takes care of XML automatically, the application may never need to use an *XMLControl* directly, but it is available if necessary. Utilities exist to convert the data model directly into a byte array or std::string as well, without needing using an *XMLControl* directly:

```
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 ownership over the builder's data
six::ComplexData* sicdData = cdb.steal();

// Add the complex data object to a SICD Container so we can write later
six::Container* container = new six::Container(six::TYPE_COMPLEX);
container->addData( sicdData );
```

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. The C++ *Container* object models the file format container, encapsulating one or more Data items in the order that they will be written (or that they were read). 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 *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 differences exist among derived *ReadControl*s. 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 monochrome 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 the 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 for the request:

```
// 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 a specified number of bytes from the stream. The example below shows how to write an 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 generated API documentation or, if necessary, 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 defects should be reported using the Trac issue tracker (currently 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 when describing the problem. Please also provide sample code. If the code in question 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. Otherwise please post an issue on the Trac under the category "help." Please check the FAQ page before posting.

Notes

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