Version

2.1

JEFFERSON LAB

Data Acquisition Group

EVIO User's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

EVIO User's Guide

Elliott Wolin
wolin@jlab.org
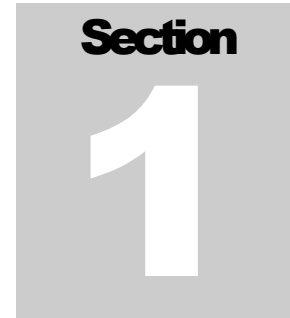

18-Jan-2007

Table of Contents

Section

1

1. Introduction

Version 1 of the CODA EVIO package, written in C, has been in use at Jefferson Lab for over a decade. It has seen extensive use in Halls A and C, where the raw data is written to disk in EVIO format, and has seen limited use in the Hall B, where PRIMEX and the GlueX BCAL test stored their raw data in EVIO format (CLAS stores raw data in BOS/FPACK format).
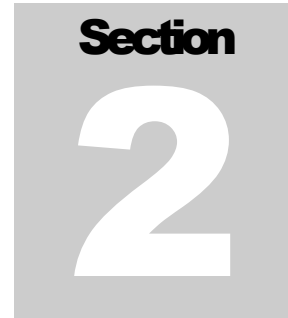
Recently the JLab DAQ group has begun upgrading and extending the EVIO package to meet the expanding needs of current and future experiments. First added were XML conversion and other utilities, support for all 1, 2, 4, and 8-byte data types, addition of a new TAGSEGMENT bank type, support for gzipped files and pipes (courtesy of Steve Wood), elimination of obsolete data types, as well as a number of bug fixes and performance enhancements.

With the advent of object-orientation and C++ we decided to perform a major upgrade to the EVIO package beyond simple wrapping of existing C code in C++. Since an EVIO event maps to a directed acyclic graph or tree, a fact which allowed us to write the XML conversion utilities, we based the object-oriented extension on the XML notion of stream and Document Object Model (DOM) parsing and DOM trees. Note that banks in an EVIO event can either be container nodes or leaf nodes, i.e. they can contain either other banks OR data, but not both (unlike XML, where a node can contain both data AND other nodes).

The object-oriented extension to EVIO described below builds upon the modern C++ standard, and makes liberal use of templates and the Standard Template Library (STL) (i.e. containers, iterators, algorithms, function objects, function object adaptors, smart pointers, etc). Fortunately users need only be familiar with a small subset of these, and examples in the tutorial below show how to do the most common tasks. Advanced users of the EVIO package should be able to take full advantage of the STL, however.

Note that the object-oriented features build upon the existing C library, and except as noted the C library continues to work as before.

Finally, the EVIO distribution can be found at http://coda.jlab.org under Downloads. The Doxygen doc is in the /html directory and is included the tar files in the /doc dir.

2. Basics of C and C++ I/O

Important: all symbols in the EVIO C++ library reside in the "evio" namespace.

2.1.   Original C I/O Library

The original EVIO C function library exists in its entirety, and is unchanged except for:

- Bug fixes
- Performance enhancements (swapper was rewritten)
- TAGSEGMENT bank type added
- Support for gzipped files and pipes
- Elimination of obsolete data types (VAX, packet and repeating types)
- Addition of stream parser

See Appendix A for details of the EVIO C library API, and section 3 for a description of stream parsing.

Note that EVIO packs multiple events into fixed sized blocks (default 8192 longwords, fixed length header) before writing the blocks to disk, and that events are split across blocks as needed. The block structure is completely hidden from the user since evRead() and evWrite() deal with single events, not blocks. The default block size can be changed with evIoctl(), which must be called immediately after evOpen(). Endian swapping is handled automatically for all but the "unknown" content type (see below).

EVIO bank structures and contentType codes are described in Appendix C. Note that files containing the obsolete data types may not be readable by the EVIO Version 2 library (e.g. VAX 32-bit floats will not be swapped properly).

Also note that the C library can always be called from C++ as long as extern "C" is used appropriately.

2.2. evioChannel

The foundation for I/O in the C++ object-oriented version of EVIO is the notion of an EVIO channel, an abstract or pure virtual class that includes the methods open(), read(), write(), and close(). Real or concrete channels extend the evioChannel class and use their constructors to supply information needed to access the underlying EVIO data stream.

There are three flavors of the write() method implementing output 1) from the internal evioChannel buffer, 2) from a user-supplied buffer, and 3) from the internal buffer in another evioChannel object. Additional methods include getBuffer() and getBufSize().

2.3. evioFileChannel

The evioFileChannel class is a subclass of evioChannel that implements I/O to and from files or file-like entities (e.g. pipes). The constructor accepts a file name, an optional mode string (default is "r"), and an optional internal buffer size (default is 8192 longwords).   The internal buffer is allocated automatically. The ioctl() method can be used to set the EVIO file block size in write mode (default is 8192 longwords), and must be called immediately after the open() method (ioctl is ignored for read mode).

This class is little more than an object-oriented wrapper around the original C function library. See the C++ tutorial for an example of how to use evioFileChannel, the C Library API in Appendix A for additional information, or the Doxygen docs for full API information.

2.4. evioEtChannel

The evioEtChannel class has not been written yet (Jan 2007…ejw). Its purpose is to read/write EVIO events to and from ET systems.

2.5. evioCMSGChannel

The evioCMSGChannel class has not been written yet (Jan 2007…ejw). Its purpose is to read/write EVIO events to and from the cMsg system
.

3. EVIO Stream Parser

Stream parsing an EVIO event involves making a single pass through the event and dispatching to user-supplied callbacks as each new bank is reached. Two versions are supplied, a C version and a C++ version. See Appendix B for details.

In the C version the user supplies two callbacks to evio_stream_parse() along with a pointer to the event buffer. evio_stream_parse() works its way through all the banks in the event in order, calling the callbacks as each new node or bank is reached. One callback is called when container banks (ones containing other banks, not data) are reached, the other when leaf or data banks are reached.

In the C++ version the evioStreamParser constructor is given an evioChannel object containing an event (e.g. an evioFileChannel object which obtained an event via its read() method) and a user-written callback handler object. The latter implements two methods: containerNodeHandler() and leafNodeHandler(). containerNodeHandler() is called when a container node or bank is reached, and leafNodeHandler() is called when a leaf or data node is reached.

In general, stream parsing may be useful for a quick pass through the data, but in C++ DOM parsing and DOM trees (see the next section) are the preferred ways to deal with EVIO events in all but the simplest cases.

4. EVIO DOM Parser and DOM Trees

In analogy with XML DOM parsing, the EVIO DOM parser constructs an in-memory object-oriented representation of an EVIO event.   This in-memory representation is stored as an instance of the evioDOMTree class. The evioDOMTree constructor can automatically construct the tree based on an event contained in an evioChannel object (e.g. an instance of evioFileChannel). Manual construction and modification of trees is also possible.

The tree itself consists of a hierarchy of nodes of two types, container nodes and leaf nodes. Container nodes hold lists of other nodes; leaf nodes contain vectors of data. Both node types inherit from the abstract base class evioDOMNode. The top node in the tree is called the root node. Note that the API is defined entirely by the evioDOMNode class, and that user code never calls its sub-classes directly.

4.1.   evioDOMNode

This is the abstract base class for the two concrete node types described above, and the only class that users deal with directly. The evioDOMNode class contains a parent pointer, parent tree pointer, tag, num, and content type. The latter three correspond to the fields in EVIO bank headers in EVIO files. Legal content types are listed in Appendix C.

Nodes are created via static factory methods :

```
evioDOMNodeP evioDOMNode::createEvioDOMNode()
```

where evioDOMNodeP is a node pointer and all objects are created on the heap. Other methods include toString(), which returns an XML fragment representing the node; bool isContainer() and isLeaf(), and a few others described below. operator== and operator!= are defined to compare tags if the argument is an integer, or tag and num if the argument is a tagNum pair (see API docs).

4.1.1.   getChildList()

getChildList() returns a pointer to the child list of an evioDOMNode that actually is a container node. NULL is returned if the node is a leaf node. See the tutorial for more details.

4.1.2.   geVector<T>()

getVector<T>() returns a pointer to the data vector contained in a leaf node of type T, where T is one of the many supported data types (int, unsigned int, double, etc). NULL is returned if the node is a container node, or if it is a leaf node containing a different type. See the tutorial for more details.

4.2.   evioDOMTree

This class represents the EVIO DOM tree or event in memory. It contains a pointer to the evioDOMNode that forms the root of the tree (type is always BANK), and the name of the tree (default is "evio"). It can construct a tree from an event contained in an evioChannel object (see the tutorial). Manual construction of a tree is discussed below.

Methods include toString(), which returns an XML string representing the entire contents of the tree, and getNodeList(Predicate P), which returns a (pointer to a) list of pointers to all nodes in the tree satisfying the predicate P. See the C++ tutorial or the API docs for details.

4.2.1.   Manual evioDOMTree construction

Manual construction of an evioDOMTree might typically happen in a Monte Carlo program that outputs simulated data. A root node must first be created, then it can be filled with either data if it is a leaf node, or pointers to other evioDOMNode objects if it is a container node. This process can be repeated recursively until a complete tree is formed. Then e.g. the tree can be written to a file via use of the write() method of an evioFileChannel object.

evioDOMTree constructors exist that can automatically create the root node. Alternatively, you can create the root node yourself and supply it to the tree constructor directly.

Nodes are created via the static factory methods evioDOMNode::createEvioDOMNode(), and can be added to the root node (assuming it is a container) or other container nodes in

a variety of ways. See the tutorial for examples of how to create nodes, add nodes to the child lists of container nodes, and add data to leaf nodes.

4.2.2.   Modification of existing trees

Modification of an existing tree might typically happen in a reconstruction program that first constructs an evioDOMTree from data read in by an evioFileChannel object, and then adds additional reconstructed data to the tree before writing it out again. The program might create one or more sub-trees containing the new data, then add the subtrees to the child lists of container nodes in the original tree. Further, sub-trees of the existing tree might be deleted by removing them from the child lists of container nodes, or moved from one container node to another.

These operations are easily carried out via the evioDOMNode methods cut(), cutAndDelete(), and move(). See the tutorial for details.

Section

5

5. Java I/O

The current Java library only implements the functionality of the original EVIO C library. We have not implemented stream or DOM parsers or DOM trees in Java yet (Oct 2005…ejw). We plan to do this when the C++ API is in final form, as the Java API will mimic the C++ API.

See the Java tutorial below, or the Javadoc for details.

6. Utilities

The utilities described below can be used to convert from binary EVIO to ASCII XML format and back, and to selectively copy EVIO events from one binary file to another. Below the term "event tag" refers to the tag of the outermost bank in an event, which is always of type BANK (two-word header, includes num).

6.1. evio2xml

evio2xml is a flexible utility that reads a binary EVIO file and dumps selected events in XML format to stdout or to a file:

```
$ evio2xml –h

   evio2xml [-max max_event] [-pause] [-skip skip_event]
            [-dict dictfilename]
            [-ev evtag] [-noev evtag] [-frag frag] [-nofrag frag]
            [-max_depth max_depth]
            [-n8 n8] [-n16 n16] [-n32 n32] [-n64 n64]
            [-w8 w8] [-w16 w16] [-w32 w32] [-w64 w64]
            [-verbose] [-xtod] [-m main_tag] [-e event_tag]
            [-indent indent_size] [-no_typename] [-debug]
            [-out outfilename] [-gz] filename
```

where most options customize the look and feel of the XML output, and defaults should be satisfactory. –max specifies the maximum number of events to dump, –pause causes evio2xml to pause between events, -skip causes it to skip events before starting to dump them, -out tells it to send the output to outfilename instead of stdout, and -gz gzips the output file. –ev can be used multiple times to select event tags of events to be dumped, and –noev the same but to exclude event tags. Similarly, -frag and –nofrag can be used to allow or exclude dumping of banks with specific internal (not event) bank tags. By default the bank tags are printed as numbers. The user can specify ASCII strings to be used instead in a tag dictionary (via –dict). Contact the DAQ group to get an example dictionary file.

6.2. *xml2evio*

xml2evio converts an EVIO XML file to a binary EVIO file:

```
$ xml2evio -h

   xml2evio [-xml xmlfilename] [-max max_event] [-skip nskip]
            [-evio eviofilename] [-dict dictfilename]
            [-m main_tag] [-e event_tag]
```

where –xml specifies the input file name, –max specifies the maximum number of events to convert, -skip causes xml2evio to skip events before converting, -evio specifies the output file name, -dict is as described above for evio2xml, and –m and –e handle custom XML main and event tags.

6.3. *eviocopy*

eviocopy copies selected events from a binary EVIO file to another binary EVIO file.

```
$ eviocopy -h

   eviocopy [-max max_event] [-skip skip_event]
            [-ev evtag] [-noev evtag] [-debug]
            input_filename output_filename
```

where –max specifies the maximum number of events to copy, -skip cause eviocopy to skip events, -ev causes eviocopy to only copy events with the specified event tag, and -noev inhibits copying of events with the specified tag. –ev and –noev can be specified multiple times on the command line.
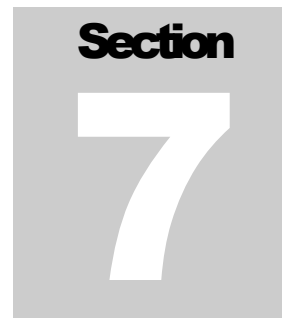
7. Future Plans

We have received many suggestions for improvement, and more are welcome. In no particular order:

- Event search facility
- High-speed random-access I/O capability
- AIDA-compliant interface
- Support compound leaf data types
- Auto-serialization of structs or objects

Please let us know if any of the above, or any other feature, is important for your work.

Section

8

8. C++ Tutorial

Below are examples showing: how to read an event from a file into an evioDOMTree; how to query the tree to get lists of node pointers that satisfy various criteria and how to work with the lists; and how to modify the tree. Some advanced topics follow.

8.1 Simple event I/O

Below is a simple example that uses an evioFileChannel object to open and read an EVIO file, then create an evioDOMTree from the event in the evioFileChannel object, then dump the event to stdout:

```cpp
#include <evioUtil.hxx>

using namespace evio;
using namespace std;


int main(int argc, char **argv) {

  try {

    // create evio file channel object for reading, argv[1] is filename
    evioFileChannel chan(argv[1], "r");

    // open the file
    chan.open();

    // loop over events
    while(chan.read()) {

      // create tree from contents of file channel object
      evioDOMTree tree(chan);

      // print tree
      cout << tree.toString() << endl;
    }

    // eof reached...close file
    chan.close();

  } catch (evioException e) {
```

```
    cerr << endl << e.toString() << endl << endl;
    exit(EXIT_FAILURE);
  }

  // done
  exit(EXIT_SUCCESS);
}
```

The tree can be written to a file via the write() method of the evioChannel class.

8.2   Querying the event tree

There are many ways to query an evioDOMTree to get lists of subsets of nodes in the tree. To get an STL list of pointers to all nodes in the tree:

```
    evioDOMNodeListP pList = tree.getNodeList();
```

(Note to experts: evioDOMNodeListP is actually auto_ptr< list<evioDOMNodeP> >, where evioDOMNodeP is evioDOMNode*)

Here no predicate is given to getNodeList() so all pointers are returned. To get a list of pointers to just container nodes:

```
    evioDOMNodeListP pContainerList = tree.getNodeList(isContainer());
```

where isContainer() is a function object provided with the EVIO package (see Appendix D for a list of all supplied function objects). Similarly, to get a list of just leaf nodes:

```
    evioDOMNodeListP pLeafList = tree.getNodeList(isLeaf());
```

To get a list of pointers to nodes satisfying arbitrary user criteria:

```
    evioDOMNodeListP pMyList = tree.getNodeList(myChooser);
```

where myChooser() is a simple C function instead of a function object. An example that specifies particular tag/num combinations is:

```
bool myChooser(const evioDOMNodeP node) {
  return(
          ((node->tag==3)&&(node->num==0)) ||
          ((node->tag==2)&&(node->num==1))
          );
}
```

To print all the nodes in the list (there are many ways to do this):

```
    for_each(pList->begin(), pList->end(), toCout());
```

for_each() is one of a large number of STL algorithms. It accepts an STL iterator range (pList->begin(), pList->end()) and applies the function object in its third argument to

each object in the iterator range in turn. Here toCout() is another of the many function objects supplied by the EVIO package. toCout() invokes the toString() method of the objects pointed to by the iterator, then streams the result to cout.

To print just leaf nodes, this time using iterators:

```
evioDOMNodeList::iterator iter;
for(iter=pLeafList->begin(); iter!=pLeafList->end(); iter++) {
   cout << endl << (*iter)->toString() << endl;
}
```

Note that (*iter) is an evioDOMNodeP, i.e. a pointer to an evioDOMNode object.

To count the number of leaf nodes with tags between 0 and 20 (this is an inefficient algorithm shown for illustration only):

```
for(int tag=0; tag<=20; tag++) {
   cout << "There are "
        << count_if(pLeafList->begin(), pLeafList->end(), tagEquals(tag))
        << " leaf nodes with tag " << tag << endl;
}
```

count_if() is another STL algorithm that counts all objects within the iterator range for which the predicate in the third argument is true. tagEquals() is another EVIO function object that returns true if the tag of the object pointed to by the iterator is equal to the argument given to the tagEquals() constructor, in this case the loop index "tag".

To search the full list and print the data from all leaf nodes containing floats (i.e. vector<float>) using the evioDOMNode member function getVector():

```
evioDOMNodeList::iterator iter;
for(iter=pList->begin(); iter!=pList->end(); iter++) {

vector<float> *v = (*iter)->getVector<float>();
   if(v!=NULL) {
      cout << endl << endl << "Float node data:" << endl;
      for(int i=0; i<v->size(); i++) cout << (*v)[i] << endl;
   }
}
```

Note that getVector<T>() returns NULL if the node is not a leaf node containing (in this case) floats. You can tell what type of data is contained in a node via the getContentType() member function. See Appendix C for a list of legal content types.

To search the full list and access the child lists of container nodes using getChildList():

```
evioDOMNodeList::iterator iter;
for(iter=pList->begin(); iter!=pList->end(); iter++) {

   evioDOMNodeList *pChildList = (*iter)->getChildList();

   cout << "Node has " << pChildList->size() << " children" << endl;
```

```
    if(pChildList->size()>0) {
       evioDOMNodeList::const_iterator cIter;
       for(cIter=pChildList->begin(); cIter!=pChildList->end(); cIter++) {
         cout << "child has tag: " << (*cIter)->tag << endl;
       }
    }
 }
```

8.3 Manipulation of the event tree

To add a new leaf node containing integers to the root node (must be container) of a tree:

```
    unsigned short tag;
    unsigned char num;
    vector<int> myIntVec(100,1);

    tree.addBank(tag=5, num=10, myIntVec);
```
or:
```
    tree << evioDOMNode::createEvioDOMNode(tag=5, num=10, myIntVec);
```
or:
```
    tree.root->addNode(evioDOMNode::createEvioDOMNode(tag=5, num=10, myIntVec));
```

If cn1 is a container node somewhere in the tree hierarchy you can add a new node ln2 to cn1 (here ln2 is a leaf node containing ints) via:

```
    evioDOMNodeP ln2 = evioDOMNOde::createEvioDOMNode(tag=2, num=8, myIntVec);
    cn1->addNode(ln2);
```
or:
```
    *cn1 << ln2;
```

To append more data to ln2:

```
    vector<int> myIntVec2(100,2)
    ln2->append(myIntVec2);
```
or:
```
    *ln2 << myIntVec2;
```

To replace the data in ln2 with new data:

```
    ln2->replace(myIntVec2);
```

To move ln2 from cn1 to another container node cn3:

```
    ln2->move(cn3);
```

To cut cn1 out of the tree:

```
    cn1->cut();             // just cut it out
```
or:
```
    cn1->cutAndDelete();    // also delete cn1 and all of its contents
```

8.4   Example programs

A number of annotated example programs exist in the examples directory in the EVIO distribution.   These demonstrate how to read and write files; query and manipulate event trees; create, manipulate, modify, and delete banks; etc.

8.5   Advanced topics

The following examples cover some more advanced features and topics that can be ignored by most users:

evioDOMNodeListP is a smart pointer (auto_ptr<>) that is used to ensure the memory used by the lists returned by getNodeList() is released when the lists go out of scope. While in most respects smart pointers act like normal pointers, they have some unusual assignment semantics. If one smart pointer is set equal to another, ownership of the contents is transferred, and the original loses ownership, e.g:

```
evioDOMNodeListP p1(...);    // p1 points to something
evioDOMNodeLIstP p2();       // p2 empty
p2=p1;                       // p2 points to something, p1 is now empty!!!
```

Further, smart pointers must not be stored in STL containers. See the STL documentation for more information on smart pointers and auto_ptr.

Note that if a standard shared pointer ever appears auto_ptr<> will be replaced. We decided not to use the Boost shared pointer as Boost is not part of the standard Linux distribution. We are considering incorporating a third-party shared pointer into the EVIO library if nothing else appears. Contact EJW for more information.

9   Java Tutorial

Currently (Jan-2007…ejw) the Java EVIO class library only implements reading and writing of events from EVIO files into internal buffers. Stream and DOM parsing and DOM trees are not implemented yet.

To read events from and EVIO file and process them in Java (n.b. endian swapping is handled automatically):

```
import org.jlab.coda.jevio.*;

try {
    Jevio evio = new Jevio();

    // open file
    evio.openFile("myFile.dat","r");

    // read events until eof reached
    while(!evio.endEvent) {

            JevioEvent evt = evio.read();
            int[] buf = evt.getEvtArray();

            // process event in buf
    }

} catch (Exception e) {
    System.out.println(e);
}
```

Appendix

A

A.   C Library API

In all cases below status is set to S_SUCCESS if the call succeeds.

To open a file:

```
#include "evio.h"
status = evOpen(char *filename, char *mode, int *handle)
```

where mode is "r" or "w" for read or write (case insensitive), and handle is returned and must be used in all subsequent EVIO calls. If filename doesn't exist an error is returned for reading, and a new file is created for writing (any existing file is overwritten). evOpen() automatically detects if the input file is gzipped, and unzips automatically (gzipping on output is not supported). If filename is set to "-" input is taken from STDIN or output is sent to STDOUT. Finally, if the first character of filename is set to '|' then I/O is sent to a UNIX pipe specified by the command after the '|' (i.e. popen() is called instead of fopen()).

To read one event into the user's buffer and perform all endian swapping:

```
status = evRead(int handle, unsigned int *buffer, int buflen)
```

where the user must allocate the buffer and buflen is in 4-byte longwords.   If the buffer is too short to contain the event it is truncated to fit.

To write one event from a user buffer to the file stream:

```
status = evWrite(int handle, const unsigned int *buffer)
```

where the buffer must contain a BANK (n.b. the first word of a BANK is the length of the event in the buffer minus 1).

To close the file:

```
status = evClose(int handle)
```

To reset the physical record blocksize for the file:

```
status = evIoctl(int handle, char *request, void *argp)
```

where request is either "B" or "b", and argp is address of an integer containing the new blocksize. evIoctl MUST be called immediately after evOpen() for writing, and is ignored for reading.

Appendix

B

B.   Stream Parsing

B.1   In C

To use evio_stream_parser():

```
#include "evio.h"

int handle;
unsigned int buffer[10000];
int buflen=10000;

/* open file */
status = evOpen(myFilename, "r", &handle);

/* read events */
while(evRead(handle, buffer, buflen)==S_SUCCESS) {

   /* parse event and dispatch to callbacks */
   evio_stream_parser(buffer, node_handler, leaf_handler);

}

/* close file */
evclose(handle);
```

where the node_handler callback is of type NH_TYPE, and the leaf_handler callback is of type LH_TYPE (either can be NULL):

```
typedef void (*NH_TYPE)(int length, int ftype, int tag, int type,
                        int num, int depth);

typedef void (*LH_TYPE)(void *data,
                        int length, int ftype, int tag, int type,
                        int num, int depth);
```

where length is the length of the contents of the bank, ftype is the type of bank (BANK, SEGMENT, or TAGSEGMENT) , tag is the bank tag, type defines the content type of the bank, num is defined only for the BANK type (set to 0 for SEGMENT and TAGSEGMENT), depth is the level of the bank in the tree, and data is a pointer to the array of data contained by the leaf bank (must be cast to appropriate type before accessing data).

B.2 In C++

To use the evioStreamParser:

```cpp
#include <evioUtil.hxx>

using namespace evio;
using namespace std;


int main(int argc, char **argv) {

  try {

    // create evio file channel object for reading, argv[1] is filename
    evioFileChannel chan(argv[1], "r");

    // open the file
    chan.open();

    //  create parser and node handler objects
    evioStreamParser parser;
    myHandler handler;

    // read events and parse channel internal buffer
    while(chan.read()) {
       parser.parse(chan.getBuffer(),handler,(void*)NULL);
    }

    // eof reached...close file
    chan.close();

  } catch (evioException e) {
    cerr << endl << e.toString() << endl << endl;
    exit(EXIT_FAILURE);
  }

  // done
  exit(EXIT_SUCCESS);
}
```

where:

```cpp
class myHandler : public evioStreamParserHandler {

  void *containerNodeHandler(int length, unsigned short tag,
     int contentType, unsigned char num, int depth, void *userArg) {
        return(NULL);
  }

  void leafNodeHandler(int length, unsigned short tag, int contentType,
      unsigned char num, int depth, const void *data, void *userArg) {
  }
};
```
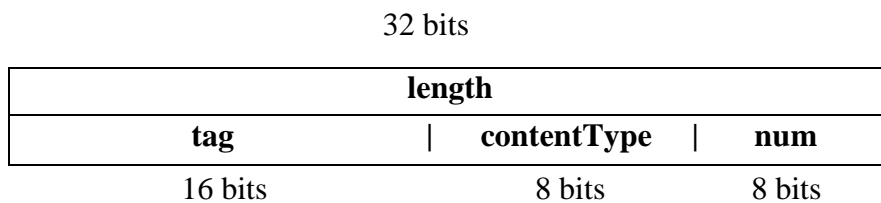
C.   EVIO Bank Structures and Content Types

EVIO data is composed of a hierarchy of banks of different types. Container banks contain other banks, and leaf banks contain an array of a single primitive data type. Three types of banks exist: BANK, SEGMENT, and TAGSEGMENT. BANK has a two-word header, the latter two have a one-word header. All banks contain a length, tag and contentType. BANK additionally has a num field. SEGMENT and TAGSEGMENT differ on the number of bits allocated to the tag and contentType. Tag and num are user-defined. contentType denotes the bank contents and the codes listed below MUST be used or endian swapping will fail. Length is always the number of 32-bit longwords to follow (i.e. bank length minus one).   Bank contents immmediately follow the bank header.
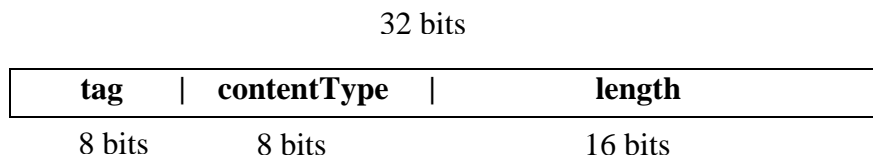
The first bank in a buffer or event MUST be a BANK. Note that the CODA DAQ system defines additional conventions for tag and num.

Bank headers for the different bank types are shown below:

BANK

32 bits

length		
tag	contentType	num

16 bits   8 bits   8 bits

SEGMENT

32 bits

tag	contentType	length

8 bits   8 bits   16 bits

TAGSEGMENT

32 bits

tag	contentType	length

12 bits   4 bits   16 bits
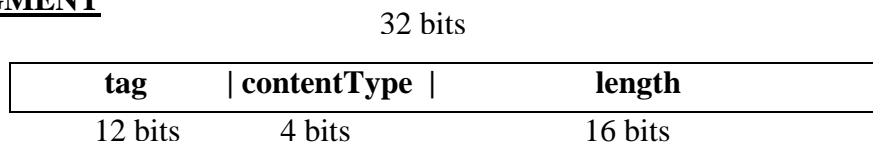
Legal contentType codes are listed below. Note that only 4-bit codes can be used in TAGSEGMENTs. Packets, repeating structures, and VAX types, which existed in EVIO Version 1, have been eliminated. Data encoded with the unknown data type is not swapped by evRead(). A string is stored as a null-terminated array of char's. Note that the 8-bit char* type is represented in DOM trees as a vector containing a single C++ string, whereas the other 8-bit char types are represented as vectors of signed or unsigned char's.

Since the bank length is always in 32-bit longwords, the number of data items stored in a bank must be calculated from the bank length and the size of the primitive data type. Data must be padded to match longword boundaries (n.b. this gives rise to a fundamental ambiguity between e.g. a bank of 11 shorts padded with an additional short and a bank with 12 shorts).

contentType	Primitive Data Type
0x0	32-bit unknown (not swapped)
0x1	32 bit unsigned long
0x2	32-bit float
0x3	8-bit char*
0x4	16-bit signed short
0x5	16-bit unsigned short
0x6	8-bit signed char
0x7	8-bit unsigned char
0x8	64-bit double
0x9	64-bit signed long long
0xa	64-bit unsigned long long
0xb	32-bit signed long
0xc	TAGSEGMENT
0xd	SEGMENT
0xe	BANK
0x10	BANK
0x20	SEGMENT
0x40	TAGSEGMENT

D. EVIO Function Objects

A number of useful adaptable function objects for applying STL algorithms to the lists returned by getNodeList() are provided. Adaptable means they can be used with STL function object adaptors (see the STL documentation). Their constructors are:

operator() returns bool:

```
typeIs<T>(void)
typeEquals(int aType)
tagEquals(unsigned short aTag)
numEquals(unsigned char aNum)
tagNumEquals(unsigned short aTag, unsigned char aNum)
parentTypeEquals(int aType)
parentTagEquals(unsigned short aTag)
parentNumEquals(unsigned char aNum)
parentTagNumEquals(unsigned short aTag, unsigned char aNum)
isContainer(void)
isLeaf(void)
```

operator() returns void:

```
toCout(void)
```

Appendix

E

E. Revision History

Version	Date	Comment
1.0	**Early 1990's**	**Original C version**
2.0pre-beta	**Mid-2002**	**XML utilities, bug fixes, I/O enhancements, etc.**
2.0beta	**Oct 2005**	**C++ API, stream and DOM parsing and trees, etc.**
2.0	Jan 2007	**Full tree manipulation API implemented**
2.1	**April 2011**	**Tag evio distribution and label as version 2.1**