

Version

4.1

JEFFERSON LAB

Data Acquisition Group

EVIO User's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

EVIO User's Guide

Elliott Wolin
wolin@jlab.org

Carl Timmer
timmer@jlab.org

1-Nov-2012

Table of Contents

| | |
|---|-----------|
| 1. Introduction..... | 5 |
| 1.1. Evio Version 1 | 5 |
| 1.2. Evio Versions 2 & 3..... | 5 |
| 1.3. Evio Version 4 | 6 |
| 1.3.1. File Format Block Size | 6 |
| 1.3.2. Network Communication Format | 6 |
| 1.3.3. Expanded User Interface | 6 |
| 1.3.4. Dictionary..... | 6 |
| 1.3.5. Padding..... | 6 |
| 1.3.6. Data Formats..... | 7 |
| 1.3.7. Random Access | 7 |
| 1.3.8. Append Mode | 7 |
| 2. Basics of the C Library..... | 8 |
| 2.1. Starting to use Evio | 8 |
| 2.2. Reading events..... | 8 |
| 2.3. Controlling I/O through <i>evIoctl()</i> | 9 |
| 2.4. Writing events..... | 10 |
| 2.5. Network Communication Format | 10 |
| 2.6. Dictionary..... | 10 |
| 2.7. Data Formats..... | 11 |
| 2.8. Documentation | 11 |
| 3. Basics of the C++ Library | 12 |
| 3.1. <i>evioChannel</i> | 12 |
| 3.2. <i>evioFileChannel</i> | 12 |
| 3.3. <i>evioEtChannel</i> | 12 |
| 3.4. <i>evioCMMSGChannel</i> | 13 |
| 4. EVIO Stream Parser | 14 |
| 4.1. in C | 14 |

| | | |
|-----------|--|-----------|
| 4.2. | <i>In C++</i> | 15 |
| 5. | EVIO DOM Parser and DOM Trees | 17 |
| 5.1. | <i>evioDOMNode</i> | 17 |
| 5.1.1. | <i>getChildList()</i> | 18 |
| 5.1.2. | <i>geVector<T>()</i> | 18 |
| 5.2. | <i>evioDOMTree</i> | 18 |
| 5.2.1. | <i>Manual evioDOMTree construction</i> | 18 |
| 5.2.2. | <i>Modification of existing trees</i> | 19 |
| 6. | Utilities | 20 |
| 6.1. | <i>evio2xml</i> | 20 |
| 6.2. | <i>xml2evio</i> | 21 |
| 6.3. | <i>eviocopy</i> | 21 |
| 7. | C++ Tutorial | 22 |
| 7.1. | <i>Simple event I/O</i> | 22 |
| 7.2. | <i>Querying the event tree</i> | 23 |
| 7.3. | <i>Manipulation of the event tree</i> | 25 |
| 7.4. | <i>Example programs</i> | 26 |
| 7.5. | <i>Advanced topics</i> | 26 |
| 8. | Jevio (Java Evio) | 27 |
| 8.1. | <i>Building</i> | 27 |
| 8.2. | <i>Documentation</i> | 27 |
| 8.3. | <i>Basics</i> | 28 |
| 8.4. | <i>Event Creating</i> | 28 |
| 8.5. | <i>Writing</i> | 29 |
| 8.6. | <i>Reading</i> | 31 |
| 8.7. | <i>Searching</i> | 34 |
| 8.8. | <i>Parsing</i> | 35 |
| 8.9. | <i>Transforming</i> | 36 |
| 8.10. | <i>Dictionaries</i> | 36 |
| A. | Evio File Format | 38 |
| A.1 | <i>Old Format, Evio Versions 1-3</i> | 38 |
| A.2 | <i>New Format, Evio Version 4</i> | 39 |
| B. | EVIO Data Format | 41 |
| B.1 | <i>Bank Structures & Content</i> | 41 |
| B.2 | <i>Changes From Previous Versions</i> | 42 |

| | | |
|-----------|---|-----------|
| B.3 | <i>Composite Data Type</i> | 43 |
| C. | EVIO Dictionary Format | 46 |
| C.1 | <i>Evio versions 2 & 3</i> | 46 |
| C.1.1 | <i>Jevio problems</i> | 47 |
| C.1.2 | <i>C++ Evio problems</i> | 47 |
| C.2 | <i>Evio version 4</i> | 47 |
| C.2.1 | <i>Behaviors</i> | 49 |
| C.2.2 | <i>Differences between C++ and Java</i> | 50 |
| D. | EVIO Function Objects | 51 |
| E. | Revision History | 52 |

1. Introduction

1.1. *Evio Version 1*

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).

1.2. *Evio Versions 2 & 3*

In the past couple of years, in EVIO versions 2 and 3 (no difference between them), the JLab DAQ group upgraded and extended the EVIO package to meet some additional needs. 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++ the DAQ group achieved 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. However, advanced users of the EVIO package should be able to take full advantage of the STL.

INTRODUCTION

Note that the object-oriented features build upon the existing C library, and except as noted the C library continues to work as before. On the Java front, the DAQ group adopted, extended, and supports Dave Heddle's jevio package.

1.3. Evio Version 4

This brings us to EVIO version 4 which can be found at <http://coda.jlab.org> under "Downloads". The following outlines the major changes that were made.

1.3.1. File Format Block Size

In previous versions, the EVIO file format had fixed-size blocks generally set to 8192 32-bit ints (32768 bytes) including a block header. EVIO banks were often split across one or more blocks. This was largely done for error recovery when using tape storage.

In version 4, since tape storage considerations are now irrelevant, each block contains an integral number of events. Users can set the nominal block size or events/block. Writing will not exceed the given limit on events/block, but each block may contain significantly less events depending on their size. The nominal block size will be exceeded in the case that a single event larger than that size is written.

1.3.2. Network Communication Format

In order to unify file and network communications, the new file format is used for both. The C library has `evOpenBuffer` and `evOpenSocket` routines to complement the traditional `evOpen` and allows reading and writing with buffers and TCP sockets.

1.3.3. Expanded User Interface

The C library contains several new read routines which differ in their memory handling. Options for the routine `evIoctl` have been expanded. Routines for dictionary handling and other purposes have been added as well.

1.3.4. Dictionary

An xml format dictionary can be seamlessly included as the first bank of a file/network format.

1.3.5. Padding

When using 1 and 2 byte data sizes (short, unsigned short, char, and unsigned char) in previous EVIO versions, there was some ambiguity. Because EVIO format dictates each bank, segment, or tagsegment must be an integral number of 32-bit ints in length,

INTRODUCTION

specifying an odd number of shorts or non-multiple of 4 number of chars meant there were extra, unused shorts or chars that the user had to keep track of externally.

In version 4, with banks and segments (not tagsegments), these unused shorts/chars or padding are tracked in the header by using the 2 highest bits of the content type. Padding can be 0 or 2 bytes for shorts and 0-3 bytes for chars. All padding operations are completely transparent to the user.

1.3.6. *Data Formats*

There is a new data format called composite data which is used by Hall B. In a nutshell it consists of a string which describes the format of the data - allowing data of mixed types to be stored together - and is followed by the data itself.

1.3.7. *Random Access*

There is a read routine which will read a particular event (say #347) directly instead of having to read the previous (346) events.

1.3.8. *Append Mode*

There is now a writing mode which will append data to the end of an existing file.

2. Basics of the C Library

EVIO bank structures and content type codes are described in Appendix C. There have been a large number of changes to the EVIO C library. Minor, non-functional changes include the following

- bug fixes
- support for gzipped files was removed
- doxygen documentation (like javadoc for C/C++) has been added

2.1. *Starting to use Evio*

The first thing a user must do is to "open" evio and obtain a handle to be used as an argument for all other evio functions. There are now 3 possibilities in the 3 open routines:

- 1) `int evOpen(char *filename, char *flags, int *handle)`
- 2) `int evOpenBuffer(char *buffer, int bufLen, char *flags, int *handle)`
- 3) `int evOpenSocket(int sockFd, char *flags, int *handle)`

The first routine is for opening a file. The "flags" argument can "w" for writing, "r" for reading, "a" for appending, or "ra" for random access. Both the appending and random access modes are new in this version of evio and are not backwards compatible. Writing a file will overwrite any existing data, while appending will add an new events to the end of a file. Reading a file will allow access to each event in the order in which it exists in the file - in other words, it is a sequential access to the events. The random access mode, on the other hand, does a preliminary scan of the file and allows reading (not writing) of selected events no matter where they are in sequence.

The second routine is for opening a buffer. It takes a pointer to a buffer as well as its length in words (32 bit ints) as the first 2 arguments. The "flags" argument is the same as for `evOpen()` as discussed in the previous paragraph.

The third is for opening evio with a TCP socket. The first argument is the socket file descriptor of a TCP socket which was created elsewhere. The "flags" argument in this case can only be "w" for writing, "r" for reading since appending or random access makes no sense when talking about a stream-oriented medium.

2.2. *Reading events*

There are now 4 routines to read events instead of the previous one.

- 1) `int evRead(int handle, uint32_t *buffer, size_t buflen)`

- 2) `int evReadAlloc(int handle, uint32_t **buffer, uint64_t *buflen)`
- 3) `int evReadNoCopy(int handle, const uint32_t **buffer, uint64_t *buflen)`
- 4) `int evReadRandom(int handle, const uint32_t **pEvent, size_t eventNumber)`

The first is the original read routine which reads an event into a user-given buffer. Its main problem is that the caller does not generally know the size of the event before reading it and therefore the supplied buffer may be too small - resulting in an error.

The second reads an event, allocating all the memory necessary to hold it with the caller responsible for freeing that memory.

The way `evio` works internally is that a file/buffer/socket is read one block at a time into an internal buffer. The third routine simply returns a pointer to the next event residing in the internal buffer - so no memory allocation or copying is done. If the data needs to be swapped, it is swapped in place. Any other calls to read routines will cause the data to be overwritten if a new block needs to be read in. Of course, no writing to the returned pointer is allowed.

Finally, the last read routine works like the 3rd read routine described in the previous paragraph in which a pointer to an internal buffer is returned to the caller. It is valid only when `evio` has been opened in random access mode and allows the caller to read only the event of interest instead of all previous events as well.

2.3. Controlling I/O through `evIoctl()`

In previous versions, the EVIO data format had fixed-size blocks generally set to 8192 32-bit ints (32768 bytes) which included a block header generally set to 32 bytes (could theoretically be larger). EVIO banks were often split across one or more blocks. This was largely done for error recovery when using tape storage.

In version 4, since tape storage considerations are now irrelevant, each block contains an integral number of events. Users can set the nominal block size and/or the number of events per block. Writing events will not exceed the given limit on events/block, but each block may contain significantly less events depending on their size. The nominal block size will be exceeded only in the case that a single event is larger than that.

The `evIoctl()` routine, shown below,

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

can be used to change the target block size for writes if `request = "B"` (case insensitive). In this case, `argp` points to a 32-bit int filled with the number of 32-bit words to set the block size to. If setting block size fails, writes can still continue with original block size value. The minimum size is 1K + 8(header) words. The routine can also change the maximum number of events/block if `request = "N"` and `argp` points to a 32-bit int filled with the number desired.

It can obtain the total number of events in a file or buffer opened for reading or writing if `request = "E"`. This includes any event written by calling `evWrite()` (which still may be contained in an internal buffer and **not** yet actually written out to the file/buffer). Note

that this option returns an unsigned 64-bit int so make sure the `argp` points to an int of that size or you'll inadvertently overwrite sections of your code.

This routine can obtain a pointer to allocated memory containing the most recently read block header if `request = "H"`. The size of the memory is `EV_HDSIZ` number of 32-bit ints and the pointer to the memory is obtained by passing its address in `argp`. This pointer must be freed by the caller to avoid a memory leak.

As in previous versions, it returns the `evio` version number in the `argp` pointer to a 32-bit int if `request = "V"`. All request strings are case insensitive. All version 4 commands to version 3 format files are ignored.

2.4. Writing events

As in previous versions there is only 1 write routine simply because the C library will only write in the new format, so no changes here:

```
int evWrite(int handle, const uint32_t *buffer)
```

However, there is a complication when writing to a buffer that one does not run into when writing to a file or socket. Unlike a file which grows as one writes or a socket that will take any amount of data, the buffer, that the caller provides to contain what is written, is of fixed size. Thus it is convenient to keep track of how much has already been written, before continuing to write more. This can be done through the following new routine:

```
int evGetBufferLength(int handle, uint64_t *length)
```

This routine returns the number of bytes currently written into a buffer when given a handle provided by calling `evOpenBuffer()`. After the handle is closed, this no longer returns anything valid.

2.5. Network Communication Format

In order to unify file and network communications, the new file format is used for both. The C library has `evOpenBuffer` and `evOpenSocket` routines to complement the traditional `evOpen` and allows reading and writing with buffers and TCP sockets.

2.6. Dictionary

An xml format dictionary can be seamlessly included as the first event when writing events to a file, buffer, or network. Refer to Appendix D for details on the format of a dictionary. To write a dictionary, simply call the following routine

```
int evWriteDictionary(int handle, char *xmlDictionary)
```

before writing any events and it will be seamlessly included as the first event in the first block. If events have already been written, an error will be returned. When reading events, simply call the following routine to get the dictionary, as a string, if it was defined:

```
int evGetDictionary(int handle, char **dictionary, int *len)
```

2.7. *Data Formats*

There is a new data format called composite data which is used by Hall B. In a nutshell it consists of an evio format string which describes the format of the data - allowing data of mixed types to be stored together - and is followed by the data itself.

2.8. *Documentation*

Besides the document you are now reading, there are doxygen docs which are essentially javadoc web pages for C/C++ code. To those unfamiliar with doxygen, programmers include specially formatted comments in the code itself which is extracted by the doxygen program and formed into web pages for view with a web browser. The user must generate these web pages by going to the top level of the evio distribution and typing either "make doc" or "scons doc". Then simply view the doc/doxygen/html/index.html file in a browser.

3. Basics of the C++ Library

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

3.1. *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()`.

3.2. *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.

3.3. *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.

3.4. *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

.

4. 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.

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.

4.1. *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) {
```

EVIO DOM PARSER AND DOM TREES

```
/* 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).

4.2. In C++

To use the `evioStreamParser`:

```
#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()) {
```


EVIO DOM PARSER AND DOM TREES

```
        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:

```
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) {
    }
};
```

5. 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.

5.1. *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 `bool 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).

5.1.1. *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.

5.1.2. *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.

5.2. *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.

5.2.1. *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.

5.2.2. *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.

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. 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.

7.1. *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:

```
#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.

7.2. 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;
    }
}
```

7.3. 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
```

7.4. *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.

7.5. *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.

8. Jevio (Java Evio)

The current Java EVIO package, called *jevio* (actually Java package `org.jlab.coda.jevio`) was originally written by Dr. Dave Heddle of CNU and was graciously given to the JLAB DAQ group for maintenance and continued development. Since it was created independently of the C++ implementation, it differs in its interface but contains most of the same functionality.

8.1. Building

The *jevio* software package uses *ant* to compile. To get a listing of all the options available to the *ant* command, run *ant help* in the *jevio* top level directory to get this output:

```
help:
[echo] Usage: ant [ant options] <target1> [target2 | target3 | ...]

[echo]      targets:
[echo]      help      - print out usage
[echo]      env        - print out build file variables' values
[echo]      compile    - compile java files
[echo]      clean      - remove class files
[echo]      cleanall   - remove all generated files
[echo]      jar        - compile and create jar file
[echo]      install    - create jar file and install into 'prefix'
[echo]                  if given on command line by -Dprefix=dir',
[echo]                  else install into INSTALL_DIR or CODA_HOME
[echo]                  if defined (in that order)
[echo]      all        - clean, compile and create jar file
[echo]      javadoc    - create javadoc documentation
[echo]      developdoc - create javadoc documentation for developer
[echo]      prepare    - create necessary directories
```

Although this is fairly self-explanatory, executing *ant* is the same as *ant compile*. That will compile all the java. All compiled code is placed in the generated *./build* directory. If the user wants a jar file, execute *ant jar* to place the resulting file in the *./build/lib* directory.

8.2. Documentation

In addition to the documentation in this file, there is javadoc which can be generated from the full source distribution of *jevio*. While not as detailed in explanation as this chapter, it will be much more complete with each publicly accessible class and method being listed.

Javadoc can be generated by going into the top level jevio directory and executing the command ***ant javadoc***. If more detail is desired, classes and methods which are not public can be seen by creating the javadoc by executing the command ***ant developdoc***. This is more suitable for a developer. The resulting javadoc is placed in the ***./doc/javadoc*** directory. Look in that directory at the file named ***index.html*** in a browser.

8.3. Basics

There are some things necessary to know before reading and writing evio format files. However, this is **not** intended to be a full evio tutorial. First, let's look at the classes which form the basis of evio data and do some basic manipulations.

Evio's container structures are banks, segments, and tagsegments. These entities are implemented with 4 different classes. At the very top level is the ***EvioEvent*** which is just a special case (subclass) of an ***EvioBank*** with dictionary and other extra data included. Banks, of course, have 2 words (8 bytes) of header followed by data. The ***EvioSegment*** and ***EvioTagSegment*** classes represent segments and tagsegments respectively, each have 1 word of header, no num value and differing amounts of tag and type data.

To get information contained in a header, call `getHeader()` with event, bank, seg, or tagseg objects. Using the returned ***BaseStructureHeader*** object, there are methods available to get & set values for content type, tag, num, length, and padding.

Events of any complexity (containing container structures) are created using the ***EventBuilder*** class. The writing of events is done through ***EventWriter*** objects, and the reading of events through ***EvioReader***. There is also a graphical viewer of events available in ***EventTreeFrame***.

In previous versions of jevio, only files could be written and read. Currently, however, evio data can be handled by buffers as well.

The next 2 sections have examples which work together. The reading example will read what the writing example produces.

8.4. Event Creating

There are 2 ways to create an evio event. Start with the simplest first -- use the ***EventBuilder*** class to do it. This takes care of all the little details and requires only the initial calling of the constructor and subsequent calling of the `addChild()` method to create an evio event. The builder will check all arguments, the byte order of added data, type mismatches between parent & child, and will set all evio header lengths automatically. The following code uses the ***EventBuilder*** to create an event (bank) of banks with 1 child which is a bank of segments. The bank of segments also has 1 child which is a segment of shorts.

JEVIO

```
// Use the EventBuilder class to create event of banks, tag=1, num=1
EventBuilder builder = new EventBuilder(1, DataType.BANK, 1);
EvioEvent event = builder.getEvent();

// bank of segments
EvioBank bankSegs = new EvioBank(2, DataType.SEGMENT, 2);
builder.addChild(event, bankSegs);

// segment of 3 shorts
EvioSegment segShorts = new EvioSegment(3, DataType.SHORT16);
short[] sdata = new short[] {1,2,3};
segShorts.setShortData(sdata);
builder.addChild(bankSegs, segShorts);

// To remove a structure
builder.remove(segShorts);
```

The second means is to call the `insert()` method of the event or its children. This method requires the event to call `setAllHeaderLengths()` at the end to make sure all the evio headers in the event have the proper lengths set. The following code does exactly what the previous example does but does not check for the issues mentioned above:

```
// Use event constructor and insert() calls
EvioEvent event = new EvioEvent(1, DataType.BANK, 1);

// bank of segments
EvioBank bankSegs = new EvioBank(2, DataType.SEGMENT, 2);
event.insert(bankSegs);

// segment of 3 shorts
EvioSegment segShorts = new EvioSegment(3, DataType.SHORT16);
short[] sdata = new short[] {1,2,3};
segShorts.setShortData(sdata);
bankSegs.insert(segShorts);

// To remove a structure
bankSegs.remove(segShorts);

// Make sure all evio headers have correct lengths
event.setAllHeaderLengths();
```

8.5. Writing

Start writing an evio format file or buffer with an ***EventWriter*** object. Simply pick among the various constructors for your medium of choice. There are optional parameters including allowing the user to chose whether to append to or overwrite any previously existing data. The user can also set the block size and number of events per block as well as specify a dictionary and data byte order among other things. Refer to the javadoc for all of the possibilities.

Below is some example code with comments showing how the writing is done. It shows how to write to both files and buffers as well as how to define a dictionary and how to create evio data. If the reader is unfamiliar with Java's ***ByteBuffer*** class, take some time to read up on it when using buffers. It will allow you to do many things.

```

// For WRITING a file or buffer
public static void main(String args[]) {

    // Define xml dictionary
    String xmlDictionary =
        "<xmlDict>\n" +
        "  <bank name=\"bank of banks\"          tag=\"1\"    num=\"1\">\n" +
        "    <bank name=\"bank of segments\"      tag=\"2\"    num=\"2\">\n" +
        "      <leaf name=\"segment of shorts\" tag=\"3\"    />\n" +
        "    </bank>\n" +
        "    <bank name=\"bank of banks\"          tag=\"4\"    num=\"4\">\n" +
        "      <leaf name=\"bank of chars\"        tag=\"5\"    num=\"5\">/>\n" +
        "    </bank>\n" +
        "  </bank>\n" +
        "  <dictEntry name=\"last bank\"          tag=\"33\"   num=\"66\">/>\n" +
        "</xmlDict>";

    // Data to write
    byte[]  byteData1 = new byte[]  {1,2,3,4,5};
    int[]   intData1  = new int[]   {4,5,6};
    int[]   intData2  = new int[]   {7,8,9};
    short[] shortData = new short[] {11,22,33};

    // Do we append or overwrite?
    boolean append = false;

    // Do we write to file or buffer?
    boolean toFile = true;

    ByteBuffer myBuf = null;

    try {
        EventWriter writer;

        if (toFile) {
            // Create an event writer to write out the test events to file
            // along with a dictionary
            String fileName = "./myData";
            File file = new File(fileName);
            writer = new EventWriter(file, xmlDictionary, append);
        }
        else {
            // Or create an event writer to write to buffer
            myBuf = ByteBuffer.allocate(10000);
            myBuf.order(ByteOrder.LITTLE_ENDIAN);
            writer = new EventWriter(myBuf, xmlDictionary, append);
        }

        // event - bank of banks
        EventBuilder builder = new EventBuilder(1, DataType.BANK, 1);
        EvioEvent event = builder.getEvent();

        // bank of segments
        EvioBank bankSegs = new EvioBank(2, DataType.SEGMENT, 2);
        builder.addChild(event, bankSegs);

        // segment of 3 shorts
        EvioSegment segShorts = new EvioSegment(3, DataType.SHORT16);
        segShorts.setShortData(shortData);
        builder.addChild(bankSegs, segShorts);
    }
}

```

```

// another bank of banks
EvioBank bankBanks = new EvioBank(4, DataType.BANK, 4);
builder.addChild(event, bankBanks);

// bank of chars
EvioBank charBank = new EvioBank(5, DataType.CHAR8, 5);
charBank.setByteData(byteData1);
builder.addChild(bankBanks, charBank);

// Write event to file
writer.writeEvent(event);

// How much room do I have left in the buffer now?
if (!toFile) {
    System.out.println("I have " + myBuf.remaining() + " bytes left");
}

// event - bank of ints
EvioEvent lastEvent = new EvioEvent(33, DataType.INT32, 66);

// Tell jevio what the data's endianness is.
// This will not swap anything now but will
// enable it to be written out properly.
// NOT necessary to call if data is big endian.
// Call this BEFORE dealing with data!
lastEvent.setByteOrder(ByteOrder.LITTLE_ENDIAN);

// Overwrite all previous data with "setIntData"
lastEvent.setIntData(intData1);
// Append data to end with "appendIntData"
lastEvent.appendIntData(intData2);

// Write last event to file or buffer
writer.writeEvent(lastEvent);

// All done writing
writer.close();
}
catch (IOException e) {
    e.printStackTrace();
}
catch (EvioException e) {
    e.printStackTrace();
}
}

```

8.6. Reading

Start reading an evio format file or buffer with a ***EvioReader*** object. Simply pick among the various constructors for your medium of choice. There is an optional parameter allowing the user to make sure the incoming block numbers are sequential. (Find out about block numbers by reading through Appendix A which describes the evio file format). There is also an optional parameter for choosing between sequential and random-access methods for reading a file.

Looking "under the hood" for a moment, the preferred, random-access (and default) method of reading a file is to use a memory-mapped `ByteBuffer` to be able to address each byte. It is much faster than using streams to read a file sequentially. However, using

a memory-mapped file is not always possible. There is a fundamental limitation built into the Java JVM which only allows indexes of arrays to be ints (and not longs). Since ints are signed 32 bit entities, and since a ByteBuffer object is backed by an array in which each byte is addressable, the result is that only files less than or equal to $2^{31} - 1$ (2.147G) bytes in size can be mapped. Files larger than that are read sequentially using streams. This version of jevio is the first to be able to read and write large files. The option exists to also read smaller files in a sequential manner but that would only slow things down. One possibly confusing part of the jevio interface is that methods that are random-access in nature are implemented using sequential reads for large files. Thus, jevio allows for a random-access style of approach even when the underlying reading mechanism is sequential. To implement this, all files and buffers are initially scanned to find the positions of each event. Note that this may take significant time for large files.

On this matter of random-access vs. sequentially oriented method calls, these are the 2 independent means of retrieving events from a file/buffer. The following are the random-access methods of the EvioReader class:

```
getEvent(int i)
parseEvent(int i)
gotoEventNumber(int i)
```

and the sequential methods:

```
nextEvent()
parseNextEvent()
rewind()
```

When mixing calls of these two categories in one application, there is no need to worry about one type interfering with the other. For example, if an application does a series of `parseNextEvent()` calls to look at a file, then doing a `parseEvent(20)` method call in the middle of the series will **not** change the sequence of the events returned by `parseNextEvent()`.

Now a word on performance. As previously mentioned, do not choose to read sequentially when reading files < 2.1 GB in size. When reading larger files, it is usually faster to use the sequential methods. The reason for that is they read in whole blocks (not individual events) at a time. If the file was written with block sizes substantially greater in size than a single event (the default when using small events), then it will be faster. The random-access methods will, on the other hand, hop to the event of interest and only read in that single event.

It's easier to give an example of code used to read a file than to explain things abstractly. Various lines show how to get and use a dictionary, read events with the sequential or random-access methods, get the total number of events, and get & print data. The code below uses many of the available evio features for reading and will read the file or buffer created in the previous section.

```
// For READING a file or buffer
public static void main(String args[]) {

    String fileName = "/home/myAccount/myData";
    File fileIn = new File(fileName);
    ByteBuffer myBuf = null;
```

```

// Do we read from file or buffer?
boolean useFile = true;

try {
    EvioReader evioReader;
    if (useFile) {
        evioReader = new EvioReader(fileName);
    }
    else {
        myBuf.flip();
        evioReader = new EvioReader(myBuf);
    }

    // Get any existing dictionary
    String xmlDictString = evioReader.getDictionaryXML();
    EvioXMLDictionary dictionary = null;

    if (xmlDictString == null) {
        System.out.println("Ain't got no dictionary!");
    }
    else {
        // Create dictionary object from xml string
        dictionary = new EvioXMLDictionary(xmlDictString);
        System.out.println("Dictionary:\n" + dictionary.toString());
    }

    // How many events in the file?
    int evCount = evioReader.getEventCount();
    System.out.println("Read file, got " + evCount + " events:\n");

    // Use "random access" capability to look at last event (starts at 1)
    EvioEvent ev = evioReader.parseEvent(evCount);
    System.out.println("Last event = " + ev.toString());

    // Print out any data in the last event.
    // In the writing example, the data for this event was set to
    // be little endian so we need to read it in that way too.
    ev.setByteOrder(ByteOrder.LITTLE_ENDIAN);
    int[] intData = ev.getIntData();
    if (intData != null) {
        for (int i=0; i < intData.length; i++) {
            System.out.println("intData[" + i + "] = " + intData[i]);
        }
    }

    // Use the dictionary
    if (dictionary != null) {
        String eventName = dictionary.getName(ev);
        System.out.println("Name of last event = " + eventName);
    }

    // Use sequential access to events
    while ( (ev = evioReader.parseNextEvent()) != null) {
        System.out.println("Event = " + ev.toString());
    }

    // Go back to the beginning of file/buffer for sequential methods
    evioReader.rewind();

}
catch (Exception e) { e.printStackTrace(); }
}

```

8.7. Searching

Most users are also interested in searching an event, a bank, a segment, or a tagsegment for various things. To this end, jevio has a couple of built in searches for ease of use. See the javadoc for the *StructureFinder* class for details. Custom searches can be done by creating filters conforming to the *IEvioFilter* interface. Simply define an *accept()* method to determine which structures to add to a returned list. Following is an example of code that uses both the built in search for banks with particular tag/num values and also a simple, user-defined search for finding *EvioSegment* type structures with odd numbered tags.

```
// Take some event (not defined here)
EvioEvent event;

// Search it for banks (not segs, tagsegs) with particular tag & num values
int tag=1, num=1;
List<BaseStructure> list = StructureFinder.getMatchingBanks(
    event, tag, num);

if (list != null) {
    for (BaseStructure bs : list) {
        System.out.println("Evio structure named \"\" +
            dictionary.getName(bs) +
            "\" has tag=1 & num=1");
    }
}

// -----
// Search for banks/segs/tagsegs with a custom search criteria
// -----

// Define a filter to select Segment structures with odd numbered tags.
class myEvioFilter implements IEvioFilter {
    public boolean accept(StructureType type, IEvioStructure struct){
        return (type == StructureType.SEGMENT &&
            (struct.getHeader().getTag() % 2 == 1));
    }
};

// Create the defined filter
myEvioFilter filter = new myEvioFilter();

// Use the filter to search "event"
list = StructureFinder.getMatchingStructures(event, filter);
if (list != null) {
    for (BaseStructure bs : list) {
        System.out.println("Evio structure named \"\" +
            dictionary.getName(bs) + \" is Segment with odd tag");
    }
}
```

Note that any bank, segment, or tagsegment structure can call *getMatchingStructures()* directly instead of through the *StructureFinder* class.

8.8. Parsing

Users have some options while parsing events. Listeners and filters may be added to an `EvioReader` to be used while events are being parsed. The previous section has a good example of how to create a filter. One such filter can be set for a reader object allowing the user to weed out events of no interest.

Jevio also has an ***IEvioListener*** interface that can be used to define multiple listeners that operate during parsing in a SAX-like manner. For each listener, simply define 3 methods to be run -- before an event is parsed, just after a structure in the event (bank, segment, or tagsegment) has been parsed, and after the entire event has been parsed. Following is an example of code that uses both a listener and a filter.

```
// Read some evio format file
EvioReader evioReader = new EvioReader(fileName);

// Get the parser which is contained in the reader
EventParser parser = evioReader.getParser();

// Define a listener to be used with an event parser
IEvioListener listener = new IEvioListener() {

    // Run this method after each bank/seg/tagseg has been parsed
    public void gotStructure(BaseStructure topStructure,
                           IEvioStructure structure) {
        System.out.println("Parsed structure of type " +
                           structure.getStructureType());
    }

    // Run this method before the event is parsed
    public void startEventParse(BaseStructure structure) {
        System.out.println("Starting event parse");
    }

    // Run this method after the event has been parsed
    public void endEventParse(BaseStructure structure) {
        System.out.println("Ended event parse");
    }
};

// Add the listener to the parser
parser.addEvioListener(listener);

// Define a filter to select everything (not much of a filter!)
class myEvioFilter implements IEvioFilter {
    public boolean accept(StructureType type, IEvioStructure struct){
        return true;
    }
};

// Create the above-defined filter
myEvioFilter filter = new myEvioFilter();

// Add the filter to the parser
parser.setEvioFilter(filter);

// Now parse some event
EvioEvent ev = evioReader.parseEvent(1);
```

8.9. Transforming

Occasionally there can arise problems with the "num" parameter defined by a `EvioBank` header but not the header of the `EvioSegment` or `EvioTagsegment`. The ***StructureTransformer*** class can be used to transform objects between these 3 classes while taking care of the troublesome num. For example:

```
// Take an existing EvioSegment
EvioSegment seg;
int num = 10;

// Turn that segment into a bank
EvioBank bank = StructureTransformer.transform(seg, num);
```

8.10. Dictionaries

This section describes how dictionaries can be used (refer to Appendix C for the format). In general it is easiest to have one global dictionary defined when manipulating evio data. For jevio this can be set in the singleton ***NameProvider*** class/object. To set this global dictionary simply do something like:

```
// Define xml dictionary String
String xmlDictString =
    "<xmlDict>\n" +
    "    <dictEntry name=\"first bank\"    tag=\"1\"    num=\"1\"/>\n" +
    "    <dictEntry name=\"second bank\"    tag=\"2\"    num=\"2\"/>\n" +
    "</xmlDict>";

// Create a dictionary object from xml String
EvioXmlDictionary dict = new EvioXmlDictionary(xmlDictString);

// Make it the global dictionary
NameProvider.setProvider(dict);
```

Once the global dictionary is set, the question is, "how is it used"? The section in this chapter for "Searching" uses the ***StructureFinder*** class and that is the case here as well. This class uses the global dictionary if defined, but another dictionary may be specified as an argument to its methods. There three methods in this class that use the dictionary as seen below:

```
// Take some event (not defined here)
EvioEvent event;

// Names to look for
String name      = "dictionaryEntry";
String childName = "childEntry";
String parentName = "parentEntry";

// Search for structures (banks, segs, tagsegs) with a particular name
List<BaseStructure> list1 = StructureFinder.getMatchingStructures(
    event, name, dict);
```

```

// Search for structures whose parent has a particular name
List<BaseStructure> list2 = StructureFinder.getMatchingParent(
    event, parentName, dict);

// Search for structures who have a child with a particular name
List<BaseStructure> list3 = StructureFinder.getMatchingChild(
    event, childName, dict);

// Print out the list of structures
if (list2 != null) {
    for (BaseStructure bs : list2) {
        System.out.println("Structure named \"" + dictionary.getName(bs) +
            "\" has a parent named " + parentName);
    }
}

```

In order to implement other types of searches, it would be relatively simple to copy the code for any of the three methods and modify it to suit.

When a file or buffer is read, it may have a dictionary in xml format associated with it. That dictionary is accessible through the *EvioReader.getDictionaryXML()* method. For convenience, the *EvioEvent* class has a place to store and retrieve an xml dictionary string by using its *setDictionaryXML()*, *getDictionaryXML()*, and *hasDictionaryXML()* methods.

The dictionary can also be used directly as an object of the *EvioXmlDictionary* class. Once an xml string is parsed into such an object (by means of its constructor), there are methods to retrieve the parsed information. These methods can obtain tag/num pairs associated with a name and vice versa. They can also obtain data types, data formats, and descriptive comments associated with either a name or tag/num pair.

```

// Define xml dictionary String
String xmlDictString =
    "<xmlDict>\n" +
    " <dictEntry name=\"me\" tag=\"10\" num=\"0\" type=\"composite\" />\n" +
    "   <description format=\"2iN(FD)\" >\n" +
    "       Any comments can go right here!\n" +
    "   </description>\n" +
    " </dictEntry>\n" +
    "</xmlDict>";

// Create a dictionary object from xml String
EvioXmlDictionary dict = new EvioXmlDictionary(xmlDictString);

// Retrieve & print info from dictionary
System.out.println("Getting stuff for name = \"me\":");
System.out.println("    tag      = " + dict.getTag("me"));
System.out.println("    num      = " + dict.getNum("me"));
System.out.println("    type     = " + dict.getType("me"));
System.out.println("    format   = " + dict.getFormat("me"));
System.out.println("    description = " + dict.getDescription("me"));

System.out.println("Getting stuff for tag = 10, num = 0:");
System.out.println("    type     = " + dict.getType(10,0));
System.out.println("    name     = " + dict.getName(10,0));
System.out.println("    format   = " + dict.getFormat(10,0));
System.out.println("    description = " + dict.getDescription(10,0));

```



A. Evio File Format

There has been a change in the format used to store evio data in files (and in this version, buffers and over TCP sockets as well). Following are descriptions of both the old format and the new.

A.1 *Old Format, Evio Versions 1-3*

Each file is divided into equal size *blocks* with each block having a header of 8, 32-bit integers or words. The fixed sized block was used as a means to recover data in case of tape storage problems. Following is a diagram of the header:

OLD BLOCK HEADER

MSB(31) LSB(0)
 <----- 32 bits ----->

| |
|----------------------|
| Block Length |
| Block Number |
| Header Length |
| Start |
| End |
| Version |
| Reserved 1 |
| Magic Number |

- The block length is number of 32 bit words in the block (including itself). Although it is adjustable, this was generally fixed for versions 1-3 at 8192 (32768 bytes).
- The block number is an id # used by the event writer.
- The header length is the number of 32 bit words in this header. In theory this too is adjustable but in practice was always 8.
- The start is the offset in words to the first event header in block relative to the start of the block.

EVIO FILE FORMAT

- The end is the number of valid words (header + data) in the block. This is equal to the block size unless it is the last block in which case it may be less.
- The version is the current evio format version number (1,2, or 3).
- The reserved 1 is unused.
- Finally, the magic number is the value 0xc0da0100 and is used to check endianness.

Following the header is the data. Often events ended up being split across one or more blocks. The start header word was used to find the beginning of the next event's header inside the block.

A.2 New Format, Evio Version 4

Each file is still divided into *blocks* with each block having a header. In the new format, to simplify things, each block contains an integral number of events which in turn means that the size of each block is **not** fixed. Following is a diagram of the new header:

NEW BLOCK HEADER

MSB(31) LSB(0)
<----- 32 bits ----->

| | |
|----------------------|----------------|
| Block Length | |
| Block Number | |
| Header Length | |
| Event Count | |
| Reserved 1 | |
| Bit info | Version |
| Reserved 2 | |
| Magic Number | |

- The block length is number of 32 bit words in the block (including itself). In general, this will vary from block to block.
- The block number is an id # used by the event writer.
- The header length is the number of 32 bit words in this header - set to 8 by default. This can be made larger but not smaller. Even though, theoretically, it can be changed, there are no means to do this or take advantage of the extra memory through the C, C++ or Java evio libraries.
- The event count is the number of events in this block - always integral. Note that: this value should not be used to parse the following events since the very first block may have a dictionary whose presence is not included in this count.
- The Reserved 1 & 2 are unused.

EVIO FILE FORMAT

- And the magic number is the value 0xc0da0100 and is used to check endianness.
- That leaves only the bit info and version word for discussion. The version is the current evio format version number (4) and takes up the lowest 8 bits. The other bits are used to store the various useful data listed below:

BIT INFO WORD

| Bit # (0 = LSB) | Function |
|-----------------|---|
| 0-7 | Version # = 4 |
| 8 | = 1 if dictionary is included (first block only) |
| 9 | = 1 if this block is the last block in file, buffer, or network transmission |
| 13-10 | type of events in block: ROC Raw = 0 Physics = 1 Partial Physics = 2, Disentangled Physics = 3, User = 4, Control = 5, Prestart = 6, Go = 7, Pause = 8, End = 9 Other = 15 |
| 31-14 | unused |

What isn't so clear yet is how events are divided into blocks. In java, some of the constructors for the EventWriter class have arguments specifying

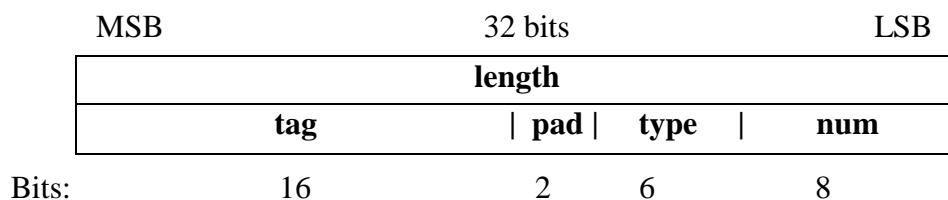
B

B. EVIO Data Format

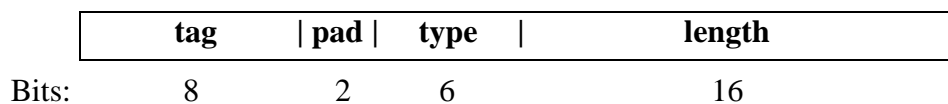
B.1 Bank Structures & Content

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 **type**. BANK additionally has a **num** field. SEGMENT and TAGSEGMENT differ on the number of bits allocated to the tag and type. Tag and num are user-defined while type denotes the bank contents and the codes listed in the table 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). New to this version of EVIO is the **pad** for both BANK and SEGMENT banks which indicates the number of bytes used for padding when type indicates 8 or 16 bit integers.

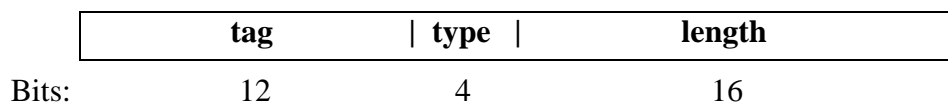
BANK HEADER



SEGMENT HEADER



TAGSEGMENT HEADER



CONTENT TYPES

| contentType | Primitive Data Type |
|--------------------|------------------------------|
| 0x0 | 32-bit unknown (not swapped) |
| 0x1 | 32 bit unsigned int |
| 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 int |
| 0xa | 64-bit unsigned int |
| 0xb | 32-bit signed long |
| 0xc | TAGSEGMENT |
| 0xd | SEGMENT |
| 0xe | BANK |
| 0xf | COMPOSITE |
| 0x10 | BANK |
| 0x20 | SEGMENT |
| 0x21 | Hollerit |

There are a few more things that the user must keep in mind:

- bank contents immediately follow the bank header
- the first bank in a buffer or event must be a BANK
- the CODA DAQ system defines specific conventions for tag and num values.

B.2 Changes From Previous Versions

There are a few changes from previous EVIO versions to take note of. A backwards-compatible change has been made for strings (type 0x3). Previously, a single ASCII, null-terminated string with undefined padding was contained in this type. Starting with version 4, an array of strings may be contained. Each string is separated by a null-termination (value of 0). A final termination of at least one 4 (ASCII char of value 4) is required in order to differentiate it from the earlier versions and to signify an end to the array. It is a self-padded type meaning it always ends on the 32 bit boundary.

EVIO DATA FORMAT

Another change is that the type of 0x40, which was redundantly defined to be a TAGSEGMENT, has been removed since its value uses bits necessary to store the padding. This is unlikely to cause any problems since it was never used.

The pad in the BANK and SEGMENT types indicates the number of bytes used for padding to 32 bit boundaries when type indicates 8 or 16 bit integers (type = 0x4, 0x5, 0x6, or 0x7). For 16 bit types pad will be 0 or 2 while for the 8 bit types it will be 0-3. Unlike previous versions, this allows EVIO to contain odd numbers of these types with no ambiguity. For example, since a bank of 3 shorts is the same length as a bank of 4 shorts (banks must end on a 32 bit boundary) previously there was no way to tell if the last short was valid data or not. Now there is. Note, however, this is **not** the case with the TAGSEGMENT bank and so it is not recommended for storing these types.

B.3 Composite Data Type

A new type - COMPOSITE - has been added which originated with Hall B but also allows for future expansion if there is a need. Basically the user specifies a custom format by means of a string. Although in practice it acts like a primitive type in that you can have a bank containing an array of them, a single COMPOSITE type looks more like 2 banks glued together. The first word comprises a TAGSEGMENT header which is followed by a string describing the data to come. After this TAGSEGMENT containing the data format string, is a BANK containing the actual data.

COMPOSITE TYPE

| tag | type | length | |
|------------------------|------|--------|-----|
| data format string ... | | | |
| length | | | |
| tag | pad | type | num |
| actual data ... | | | |

The routine to swap this data must be provided by the definer of the composite type - in this case Hall B. This swapping function is plugged into the EVIO library's swapping routine. Currently its types, tags, pad, and num values are not used. Only the lengths are significant.

There is actually another new type defined - the Hollerit type, but that is only used inside of the COMPOSITE type and refers to characters in an integer form. Following is a table of characters allowed in the data format string.

DATA FORMAT CHARACTERS

| Data format char | Meaning |
|-------------------------|--------------------------|
| (| (|
|) |) |
| i | 32-bit unsigned int |
| F | 32-bit floating point |
| a | 8-bit ASCII char |
| S | 16-bit short |
| s | 16-bit unsigned short |
| C | 8-bit char |
| c | 8-bit unsigned char |
| D | 64-bit double |
| L | 64-bit int |
| l | 64-bit unsigned int |
| I | 32-bit int |
| A | Hollerit |
| N | Multiplier in 32-bit int |

In the format string, each of the allowed characters (except ")" or "N") may be preceded by an integer which is a multiplier. Instead of trying to explain the format abstractly, let's look at the following example:

iL2(s2DNF)

This format translates into the data being read and processed in the following order: a single 32-bit unsigned int, a single 64-bit int, an unsigned short, 2 doubles, a multiplier (32 bit int), multiplier number of 32-bit floats, an unsigned short, 2 doubles, a multiplier (32 bit int), multiplier number of 32-bit floats. The data is read in according to this recipe.

There are a couple of data processing rules that are very important:

- 1) If the format ends but the end of the data is not reached, the format in the last parenthesis will be repeated until all data is processed. If there are no parentheses in the format, data processing will start again from the beginning of the format until all data is processed.
- 2) The explicitly given multiplier must be a number between 2 and 15 - inclusive. If the number of repeats is the symbol 'N' instead of a number, that multiplier will be read from data assuming 'I' format and may be any positive integer.

The COMPOSITE data type allows compact storage of different primitive data types and eliminates the need for extra banks and their accompanying headers. It does, however,

EVIO DATA FORMAT

pay a penalty in the amount of computing power needed to read, write, and swap it. For example, each time a COMPOSITE bank needs to be swapped, EVIO must read the format string, process it, and convert it into an array of ints. Then, with the converted format as a guide, EVIO must read through the data item-by-item, swapping each one. It is quite compute intensive.



C. EVIO Dictionary Format

Since names are easier for humans to deal with than pairs of numbers, the basic idea behind the dictionary is to associate a single string, a name, with 2 integers, an evio tag and an evio num. The xml protocol was used to accomplish this. The following gives the different xml formats used by the different versions of evio.

C.1 *Evio versions 2 & 3*

The xml format has been evolving. Originally, because EVIO data is stored in a hierarchical manner with banks containing banks containing data, the dictionary format was also hierarchical. In other words, a string was associated with not only the 2 numbers but a place in the hierarchy as well. The idea was that a given pair of tag/num values could occur in more than one location in the hierarchy and must be distinguishable from each other. Following is an example of the first format used:

```
<xmlDict>

  <xmldumpDictEntry name="event_1"          tag="1"          num="1"/>

  <!-- DC -->
  <xmldumpDictEntry name="DC"                tag="500"          num="0"/>
  <xmldumpDictEntry name="DC_id"             tag="500.1"        num="0.0"/>

  <xmldumpDictEntry name="DC_output"         tag="500.2"        num="0.100"/>
  <xmldumpDictEntry name="sector5"          tag="500.2.5"      num="0.100.23"/>

</xmlDict>
```

There is only one possible element - *xmldumpDictEntry*. Notice the dotted notation of the *tag* & *num* attributes. This notation, for example the tag 500.2.5, simply means that this dictionary entry has a tag value of 5, its parent has a tag value of 2, and its grandparent has a tag value of 500. Basically it is a way of specifying a place in the evio tree or hierarchy.

C.1.1 Jevio problems

The jevio-1.0 software package did **not** allow dotted notation for the tag, but did allow it for the num. The rules that jevio uses to determine whether a bank, event, segment, or tagsegment object matches a particular dictionary entry is:

- 1) if it is an EvioSegment or EvioTagSegment object, the first entry that matches its tag value is returned
- 2) if it is an EvioEvent object, the first entry that matches its tag value and the first level num value is returned
- 3) if it is an EvioBank object, the first entry that matches its tag value and the complete hierarchy of num values is returned

Although this works after a fashion, it unfortunately does not match tag values in a hierarchical manner.

C.1.2 C++ Evio problems

The C++ library's handling of the dictionary's tags & nums is not perfect either. The difficulty arises from the fact that when **creating** an evio tree of banks, segments, and tagsegments, C++ evio does not distinguish between them. Each container is simply a node that may be added, removed, cut, and pasted anywhere in the tree. Only upon serializing the tree to a file does the fact that a node is one of the 3 types come into play. In order for this model to function, all segments and tagsegments are essentially treated as banks with num = 0. Thus a dictionary entry with tag = 1 & num = 0 will match both a bank with those parameters and a segment with tag = 1 but no num. Worse yet, a node can set num = 1, be written out as a segment, and then be read back in with num = 0. This limitation must be taken into consideration when creating dictionaries & evio trees.

C.2 Evio version 4

A number changes to the previous evio dictionary format have been made. Let's start with what has been eliminated. Previously the num and tag values could be hierarchical with each level separated by a period such as:

```
tag = '1.2.3'    num = '2.5'
```

These types of values for tag and num were stored in the dictionary making the matching of a bank to a dictionary entry tricky since now the parents and children of the bank became involved. Not only was the matching complicated but a dictionary entry would have to change depending on where a particular bank was moved to in an evio event tree - very inconvenient and prone to error.

The first change eliminates these hierarchical tags & nums. Each dictionary entry is a single name associated with a single tag value and a single num value (with segments and tagsegments given a num value of 0). It becomes a simple matter to build hierarchies into the name as will be demonstrated below.

EVIO DICTIONARY FORMAT

The C/C++ dictionary and the Java dictionary now have identical formats. In the old xml format, there was only one entry type namely, the xml element of *xmldumpDictEntry*. There are now 3 types of XML dictionary elements: *dictEntry* (replaces xmldumpDictEntry which is too long), *bank* and *leaf*.

For each of these elements, the only attributes a dictionary parser will look at are *name*, *tag*, *num*, and the newly added *type* (of contained data). All other elements and attributes are ignored, so the XML can be used to define whatever else is desired. Note that only the following case-independent values are valid for type with all other values being ignored:

```
int32, uint32, long64, ulong64, short16, ushort16, char8, uchar8,  
charstar8, float32, double64, bank, segment, tagsegment, composite,  
unknown32
```

The simplest xml element is dictEntry, it just makes an entry into the map of names vs tag/num pairs:

```
<dictEntry name="fred" tag="1" num="1" />
```

Here the name "fred" is a synonym for the tag/num pair (1,1).

The elements "bank" and "leaf" are used for describing hierarchical bank structures. Take a look at the following:

```
<bank name="CLAS12" tag="1" num="0">  
  <bank name="DC" tag="20" num="0">  
    <leaf name="xpos" tag="20" num="1"/>  
    <leaf name="ypos" tag="20" num="2"/>  
    <leaf name="zpos" tag="20" num="3"/>  
  </bank>  
  <bank name="SC" tag="30" num="0">  
    <leaf name="xpos" tag="30" num="1"/>  
    <leaf name="ypos" tag="30" num="2"/>  
    <leaf name="zpos" tag="30" num="3"/>  
  </bank>  
</bank>
```

where "bank" means an evio container (bank, segment, or tagsegment), and "leaf" means an evio container with no children. The parser will generate map entries equivalent to the following:

```
<dictEntry name="CLAS12" tag="1" num="0"/>  
<dictEntry name="CLAS12.DC" tag="20" num="0"/>  
<dictEntry name="CLAS12.DC.xpos" tag="20" num="1"/>  
<dictEntry name="CLAS12.DC.ypos" tag="20" num="2"/>  
<dictEntry name="CLAS12.DC.zpos" tag="20" num="3"/>  
<dictEntry name="CLAS12.SC" tag="30" num="0"/>  
<dictEntry name="CLAS12.SC.xpos" tag="30" num="1"/>  
<dictEntry name="CLAS12.SC.ypos" tag="30" num="2"/>  
<dictEntry name="CLAS12.SC.zpos" tag="30" num="3"/>
```

EVIO DICTIONARY FORMAT

This scheme works well if all tag/num pairs are unique. That way there is a unique string associated with each tag/num pair. If multiple names are linked with the same pair, then searching for a particular name may not return the appropriate values, and searching for a tag/num pair may not return the appropriate name. Likewise, if a single name is linked with multiple pairs, the same confusion can result. In order to avoid these problems, both the C++ and Java implementations of the dictionary only allow unique mappings.

In addition to the tag, num, and name attributes, the dictionary can also hold the type information about the contents of an evio container (unknown types are ignored). For example the following associates "fred" with 32 bit signed integers:

```
<dictEntry name="fred" tag="1" num="1" type="int32" />
```

The new composite type of data requires even more information about the format of the data inside. To accommodate this, all dictionary entries may now have a *description* xml subelement defined. These descriptions may have the *format* attribute defined as well:

```
<xmlDict>
  <dictEntry name='myName' tag='123' num='456' type='composite' >
    <description format='FDNi' >
      F   TDC
      D   ADC min=5.0 max=10.0
      N   multiplier
      i   scaler bits0-15=counter1 bits15-32=counter2
    </description>
  </dictEntry>
</xmlDict>
```

The description and format can be anything meaningful to the user. Hall D will use a set format for both entries when using composite type data so they can be parsed and additional information extracted from it. This is done to allow flexibility to the user but not in a way that would be a constantly changing specification for evio.

C.2.1 Behaviors

There are a few other issues that need to be addressed. The use of the "leaf" element is optional and may be replaced by "bank". However, if "leaf" is used, it may not have any children. Even though XML is case-sensitive, in the parsing of the dictionary, all the accepted elements' and attributes' cases are ignored.

The rules that jevio now uses to determine whether a bank, event, segment, or tagsegment object matches a particular dictionary entry is:

- 1) if it is an EvioEvent, EvioBank, EvioSegment or EvioTagSegment (any kind of container) object, the first and only entry that matches its "tag" value and its "num" value is returned
- 2) if it is an EvioSegment or EvioTagSegment object, its num value is assumed to be 0 (zero) for purposes of matching

The xml representation of a dictionary can be embedded in a larger xml document. Giving this larger document as the dictionary to be parsed is perfectly acceptable and the

EVIO DICTIONARY FORMAT

code will pick out the dictionary portion. If multiple dictionaries are included, only the first is used and the rest are ignored.

Furthermore, irrelevant xml elements and attributes may be present and are simply ignored. When the `jevio toXml()` method of a dictionary is called, only the dictionary portion of the original, full xml document is returned as a String.

C.2.2 Differences between C++ and Java

There still are a couple differences between Java and C++. The first is if a "leaf" entry has any children, `jevio` will print a warning and ignore any such children while `evio` will throw an exception. Similarly, if there are duplicate entries - either names or tag/num pairs - an exception is thrown in C++ but in Java they are ignored (including any of their children) and a warning is printed. These differences are allowed for backwards compatibility purposes in `jevio`. Be aware that in `jevio` all the `dictEntry` elements are processed before the `bank` and `node` elements. So if there are duplicate entries, any `dictEntry` elements are given preference over the `bank` or `leaf`.

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)
```



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 |
| 4.0 | Aug 2012 | Random access, append mode, new file format |
| 4.1 | Nov 2012 | Read and write files > 2.1 GB |