EVIO 6.0 User's Guide

Carl Timmer

Jefferson Lab Experimental Physics Software
and Computing Infrastructure group

3-May-2021

12000 Jefferson Ave
Newport News, VA 23606
Phone 757.269.7100

Chapter 1

1. Introduction to Evio Versions

1.1 *Version 1*

Version 1 of the CODA EVIO package, written in C, was 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 stored raw data in BOS/FPACK format).

1.2 *Versions 2 & 3*

In EVIO format 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 instituted an upgrade to the EVIO package beyond simple wrapping of existing C code in C++. Since an EVIO event maps to a 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). Users need only be familiar with a small subset of C++ capabilities; however, advanced users of the EVIO package should be able to take full advantage of the STL.

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 *Version 4*

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 Splitting Files

The C, C++, and Java evio libraries all implement a means to limit the size of an evio file being written by splitting it into multiple files. There are facilities to make the automatic naming of these files simple.

1.3.5 Dictionary

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

1.3.6 First Event

Occasionally it is useful to have the same event appear in each split file, for example including a config event in each file. Such an event is called the first event since typically it is the first event after the dictionary in each of the splits. One can define a first event for each file.

1.3.7 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, 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.

1.3.8 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.

1.3.9 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.

1.3.10 Append Mode

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

1.3.11 Thread Safety

The libraries are now designed to be thread safe.

1.3.12 Graphics

The Java library has a graphical user interface for viewing evio format files.

1.4 *Version 6*

The following outlines the major changes that were made. A big factor for introducing another evio version was the desire to compress data in each block (now called a record). The HIPO format, in use in Jefferson Lab's HallB, was merged with evio along with much of the code to do (un)compression of data. This has added a great deal of complexity to the record headers (which are not compressed).

1.4.1 New Block or Record Format
Version 6 has an expanded record header which allows for bookkeeping associated with compression. It also allows for a user-defined portion of the header along with an index of the position of each contained event.

1.4.2 Compressed Data
Each record can contain data compressed in gzip, lz4 or lz4_best format.

1.4.3 Data Format Agnostic API
The HIPO classes which were incorporated into evio are data format agnostic. Thus, part of the evio library can now deal with data not in evio format.

1.4.4 New C++ library
The C++ library in this version was ported from Java. From a programmer's point of view, this allows for simpler code maintenance. The C++ library of the previous version is no longer available.

Chapter 2

2 Evio Installation

The evio library is available in three languages, C, C++, and Java. All code is contained in the github repository,

https://github.com/JeffersonLab/evio.git

The **evio-6.0** branch contains the most recent version of evio. Documentation is contained in the repository but may also be accessed at:

https://coda.jlab.org/drupal/content/event-io-evio/

2.1 *C*

The C library is called libevio. It is a library with historically limited capabilities. To compile it, follow the directions below for the C++ compilation which will include the C as well. The C++ library is much more extensive in scope.

Having said that, the C library and executables can be compiled without any C++. This can be done in 2 ways:

```
scons --C
```

or

```
mkdir build
cd build
cmake .. -DC_ONLY=1
```

For more details concerning scons and cmake, see the following section on C++.

2.2 *C++*

The C++ library is called libeviocc. The current C++ evio library is entirely different from the previous version (5.2) as it has been ported from the Java code. This was done for a number of reasons. First, a much more comprehensive C++ library was desired than was currently existing. Second, it needed major, new capabilities such as being able to (un)compress data. Third, it had to use a new format developed from the merging of Java evio version 4 and the Java HIPO library. Finally, the author and maintainer of the previous code was no longer working at Jefferson Lab. The simplest solution was to

port the well-tested Java code which avoided having to redesign complex software from scratch.

C++ evio is supported on both the MacOS and Linux platforms. C++ version 11 is used, and gcc version 5 or higher is required.

2.2.1 Prerequisites

2.2.1.1 Disruptor

Evio depends upon the Disruptor-cpp software package available from a fork of the original package at github at https://github.com/JeffersonLab/Disruptor-cpp . In terms of functionality, it is an ingenious, ultrafast ring buffer which was initially developed in Java and then ported to C++. It's extremely useful when splitting work among multiple threads and then recombining it. To build it, do this on the Mac:

```
1) git clone https://github.com/JeffersonLab/Disruptor-cpp.git
2) cd Disruptor-cpp
3) mkdir build
4) cd build
5) cmake .. -DCMAKE_BUILD_TYPE=Release
6) make
7) setenv DISRUPTOR_CPP_HOME <../>
```

If using Jefferson Lab's Redhat Enterprise 7 do:

```
1) git clone https://github.com/JeffersonLab/Disruptor-cpp.git
2) cd Disruptor-cpp
3) mkdir build
4) cd build
5) cmake .. -DCMAKE_BUILD_TYPE=Release  -DCMAKE_C_COMPILER=/apps/gcc/5.3.0/bin/gcc
   -DCMAKE_CXX_COMPILER=/apps/gcc/5.3.0/bin/g++
6) make
7) setenv DISRUPTOR_CPP_HOME <../>
```

The compilation instructions can also be found in the README file. Note that it requires GCC 5.0 / Clang 3.8 / C++14 or newer and boost. Its shared library must be installed where evio can find it. If not compiling on Jefferson Lab's RHEL7, either your default compilers must meet this criteria or you must specify the proper ones on the cmake command line.

2.2.1.2 Boost

Besides the disruptor library, evio requires the boost libraries: boost_system, boost_thread, and boost_chrono.

2.2.1.3 lz4

And finally, evio depends on the lz4 library for compressing data in the lz4 and gzip formats. If it isn't already available on your machine, it can be obtained from the lz4 repository on github.

```
1) git clone https://github.com/lz4/lz4.git
2) cd lz4
3) make
4) make install
```

2.2.2 Building

There are 2 different methods to build the C++ library and executables. The first uses **scons**, a Python-based build software package which is available at https://scons.org . The second uses cmake and make. Also, be sure you've set the DISRUPTOR_CPP_HOME environmental variable.

2.2.2.1 Scons

To get a listing of all the local options available to the scons command, run ***scons -h*** in the top-level directory to get this output:

```
-D                  build from subdirectory of package

local scons OPTIONS:
--C                 compile C code only
--dbg               compile with debug flag
--32bits            compile 32bit libs & executables on 64bit system
--prefix=<dir>      use base directory <dir> when doing install
--incdir=<dir>      copy header files to directory <dir> when doing install
--libdir=<dir>      copy library files to directory <dir> when doing install
--bindir=<dir>      copy binary files to directory <dir> when doing install
install             install libs, headers, and binaries
install -c          uninstall libs, headers, and binaries
doc                 create javadoc (in ./doc)
undoc               remove javadoc (in ./doc)
tar                 create tar file (in ./tar)

Use scons -H for help about command-line options.
```

Although this is fairly self-explanatory, executing

```
1) use gcc/5.3.0   # if on CUE system with Redhat 7
2) cd <evio dir>
3) scons install
```

will compile and install all the code. By default, all libraries, executables and includes are installed under the directory given by the **CODA** env variable. If the command line options –prefix, --incdir, --libdir, or –bindir are used, they take priority.

To compile a debug version, execute:

```
scons install --dbg
```

2.2.2.2 Cmake

Evio can also be compiled with cmake using the included CMakeLists.txt file. To build the C and C++ libraries and executables on the Mac:

```
1) cd <evio dir>
2) mkdir build
3) cd build
4) cmake .. -DCMAKE_BUILD_TYPE=Release
5) make
```

If on redhat 7 linux this will be:

```
1) cd <evio dir>
2) mkdir build
3) cd build
4) cmake .. -DCMAKE_BUILD_TYPE=Release  -
   DCMAKE_C_COMPILER=/apps/gcc/5.3.0/bin/gcc  -
   DCMAKE_CXX_COMPILER=/apps/gcc/5.3.0/bin/g++
5) make
```

To build only C code, place –DC_ONLY=1 on the cmake command line. In order to compile all the examples as well, place –DMAKE_EXAMPLES=1 on the cmake command line.

The above commands will place everything in the current “build” directory and will keep generated files from mixing with the source and config files.

In addition to a having a copy in the build directory, installing the library, binary and include files can be done by calling cmake in 2 ways:

```
1) cmake .. -DCMAKE_BUILD_TYPE=Release -DCODA_INSTALL=<install dir>
2) make install
```

or

```
1) cmake .. -DCMAKE_BUILD_TYPE=Release
2) make install
```

The first option explicitly sets the installation directory. The second option installs in the directory given in the CODA environmental variable. If cmake was run previously, remove the CMakeCache.txt file so new values are generated and used.

To uninstall simply do:

```
make uninstall
```

2.2.3 Documentation

The documentation for this software has already been created and is hosted at https://coda.jlab.org/drupal/content/event-io-evio/ (which, perhaps, you are reading right now).

2.2.3.1 Doxygen

All the source code contains doxygen style comments. One may generate the associated documentation by executing

```
scons doc
```

in the top-level directory. The result can be viewed by using your web browser to view the created doc/doxygen/C/html/index.html and doc/doxygen/CC/html/index.html files. To regenerate these files, call

```
scons undoc
scons doc
```

For those who wish to fine-tune the generated files, the doxygen configuration files, **doc/DoxyfileCC** and **doc/DoxyfileC**, can be modified to suit. The doxygen comments can be viewed from the github pages website as well so explicity dealing with doxygen is not necessary.

Now the documentation can also be generated by using cmake (assuming it's been run previously at least once):

```
1)  cd <evio dir>/build
2)  cmake --build . --target docCC docC
```

2.2.3.2 User's Guide

The user's guide (the document you're reading right now) is written and stored as a word document in the doc/users_guide directory. Of course, this can be viewed in Microsoft Word. For user's convenience, it's also available in pdf format. Transforming this document into something that can be hosted on a website can be done in a number of different ways.

2.3 *Java*

The current Java evio 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. A large amount of additional work has been done since that time. As previously mentioned, evio now uses a new format developed from the merging of evio version 4 and the HIPO library.The code will compile using Java version 8 or later.

2.3.1 Prerequisites

To begin with, the jar files necessary to compile an evio jar file are in the **java/jars** directory. They are compiled with Java 8. There are 2 subdirectories: 1) java8, which contains all such jars compiled with Java 8, and 2) java15 which contains all jars compiled with Java 15.

Evio depends upon the LMAX-Exchange/disruptor software package available from github whose fork is at https://github.com/JeffersonLab/disruptor . In terms of functionality, it is an ingenious, ultrafast ring buffer which was initially developed for use the in the commodities exchange markets. It's extremely useful when splitting work among multiple threads and then recombining it. Although there are many branches in this git repository, only 2 branches have had the necessary changes to be compatible with CODA. These are the **master** and **v3.4** branches. The v3.4 branch should be compiled with Java 8 (it does not compile with Java 15) while the master requires at least Java 11.

The disruptor software is provided in the **java/jars/disruptor-3.4.3.jar** file, compiled with Java 8. However, to generate this file yourself, get the disruptor software package by simply doing the following:

```
1) git clone https://github.com/JeffersonLab/disruptor.git
2) cd disruptor
3) git checkout v3.4
4) ./gradlew
```

The resulting disruptor jar file, **disruptor-3.4.3.jar,** will be found in the disruptor package's build/libs subdirectory.

One can also use the master branch which needs to be compiled with Java version 11 or greater and produces **disruptor-4.0.0.jar**. Currently this has been created with java15 and is in the **java/jars/java15** directory. Here is how to generate it:

```
1) git clone https://github.com/JeffersonLab/disruptor.git
2) cd disruptor
3) ./gradlew
```

The resulting jar will be in build/libs as before.

A jar file used in lz4 data compression, **lz4-java-1.8.0.jar** is accessible in the **java/jars** directory (compiled with Java 8). Although this is available in various versions and locations on the web, one can generate this from its source which is the **lz4/lz4-java** repository on github:

```
1) git clone https://github.com/lz4/lz4-java.git
2) cd lz4-java
3) ant ivy-bootstrap
4) ant submodule init
```

```
5) ant submodule update
6) ant
```

Generated jar files will be in **dist** subdirectory.

Another jar file, **AHACompressionAPI.jar**, also in the the java/jars directory, is for use in Compressor.java when using the AHA374 FPGA data compression board for gzip compression in hardware. This is an effort that never took off since LZ4 compresssion was so much more efficient. Thus, it may be safely ignored or removed.

2.3.2 Optimization

Optionally, one can use proguard to optimize the evio jar file. This will shrink, optimize, and improve the performance of the evio jar and combine it with the lz4, disruptor, and AHACompressionAPI jars at the same time. **For some reason, this no longer seems to work so its up to the user to debug things.** One may need to edit the myconfig.pro file. Yet, here are the steps to do it, just make sure you have gradle installed on your system:

```
1) git clone https://github.com/Guardsquare/proguard-core.git
2) cd proguard-core
3) gradle clean assemble
4) cd ..
5) git clone https://github.com/Guardsquare/proguard.git
6) cd proguard
7) ./gradlew --include-build=../proguard-core assemble
8) cd ../evio-6.0/build/lib
9) java -jar ../../../proguard/lib/proguard.jar @../../myconfigfile.pro
```

Note, the last command will only work if the evio, proguard and proguard-core directories are all in the same directory as each other. If they aren't, you must edit <evio top dir>/myconfigfile.pro file to reflect the current directory structure. The resulting optimized jar will be written as ***<evio top dir>/build/lib/jevio-6.0.optimized.jar*** or to what it is edited to be.

2.3.3 Building

The java evio uses ***ant*** to compile. To get a listing of all the options available to the ant command, run ***ant help*** in the evio 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 CODA if defined
[echo]        uninstall  - remove jar file previously installed into 'prefix'
[echo]                     if given on command line by -Dprefix=dir',
[echo]                     else installed into CODA if defined
[echo]        all        - clean, compile and create jar file
[echo]        javadoc    - create javadoc documentation
[echo]        developdoc - create javadoc documentation for developer
[echo]        undoc      - remove all javadoc documentation
[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. The **java** command in the user's path will be the one used to do the compilation.

2.3.4 Documentation

The documentation for this software has already been created and is hosted at https://coda.jlab.org/drupal/content/event-io-evio/ (which, perhaps, you are reading right now).

2.3.4.1 Javadoc

All the source code contains javadoc style comments. One may generate the associated documentation by calling

```
ant javadoc
```

in the top-level directory. If more detail is desired, classes and methods which are not public can be seen by, instead, executing the command

```
ant developdoc
```

This is more suitable for a developer. In either case, the resulting javadoc can be accessed by viewing the **doc/javadoc/index.html file** in a web browser. To regenerate this file, call

```
ant undoc
ant Javadoc
```

2.3.4.2 User's Guide

See the section above for info on the user's guide, which you are reading right now.

Chapter 3

3 Basics of the C Library

The C evio library is the original evio library. However, compared to the C++ and Java libraries, it is kept simpler and more primitive. Features such as adding dictionaries and first events to the file/buffer are not available. On the other hand, it contains no dependencies on external libraries.

When using C routines, it is entirely up to the user to provide buffers of data (events) in the exact evio format required. Thus, it requires a great deal of expert knowledge. What may help is that the evio file format is described in chapter 10, bank structures & content type are described in chapter 11, and the dictionary format is described in chapter 12.

3.1 *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, "ra" for random access, or “s” for splitting the file while writing. Writing a file will overwrite any existing data, while appending will add 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. When writing large amounts of data, it is often convenient to split the output into a number of files. This is supported by specifying the “s” flag. By proper specification of the filename

argument and by using the evIoctl() function, these split files can be automatically named.

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 with the exception of “s” since splitting makes no sense for buffers.

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 splitting, appending or random-access makes no sense when talking about a stream-oriented medium.

3.2 *Reading events*

There are now 4 routines able to read an event:

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.

3.3 *Writing events*

As in previous versions there is only 1 write routine simply because the C library will only write in the new format:

int evWrite(int handle, const uint32_t *buffer)

However, there is a complication when writing to a buffer that does not occur 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, an error can be returned if the amount of data written exceeds the buffer size; therefore, 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.

3.3.1 Splitting files

When writing significant amounts of data to a single file, that file can get very large – too large. Historically, run control was able to split the data into multiple files with an automatic file naming system. For this version of evio, the ability to split files is built in as is the naming system. Start by setting the “flags” parameter in the evOpen() call to “s”. In addition to that, the user may choose the number of bytes at which to start writing to a new file by a call to evIoctl() (see below). If not explicitly set, the split occurs at 1GB. The split files are named according to the automatic naming system whose details are given in the next section.

3.3.2 Naming files

When splitting files, a base filename is passed to evOpen() and may contain characters of the form **$(env)** where “env” is the name of an environmental variable. When a file is created, all such contructs will be substituted with the actual environmental variable’s value (or nothing if it doesn’t exist).

Similarly, the base filename may contain constructs of the form **%s** which will be substituted with the actual run type’s value (if set with evIoctl) or nothing if run type is null or was not set.

Generated files names are distinguished by a split number which starts with 0 for the first file and is incrementing by 1 for each additional file. Up to 2, C-style integer format specifiers (such as %03d, or %x) are allowed in the base filename. If more than 2 are found, an error is returned. If no "0" precedes any integer between the "%" and the "d" or "x" of the format specifier, it will be added automatically in order to avoid spaces in the generated filename. The first specifier will be substituted with the given run number value (set in evIoctl()). The second will be substituted with the split number. If no specifier for the split number exists, it is tacked onto the end of the file name.

Below is an example of how the file naming and splitting is done. Given the list of values below

```
int split          = 100000000; // split at 100MB
int runNumber      = 1;
char *runType      = "myExperiment";
char *directory    = "/myDirectory";
char *baseFilename = "my$(BASE_NAME)_%s_%x_%03d.ext";
```

and a BASE_NAME environmental variable of the value “File”, the following happens. The baseFilename string will have the environmental variable, BASE_NAME, substituted in the obvious location along with the runType substituted for the %s, the runNumber substituted for the %x (hex format), and the split number substituted for the %03d. The first 3 split files will have the names:

```
myFile_myExperiment_1_001.ext
myFile_myExperiment_1_002.ext
myFile_myExperiment_1_003.ext
```

3.4 *Controlling I/O through evIoctl()*

Some control over evio settings is given to the user with the evIoctl() routine, shown below,

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

It can be used, for example, to set the target block size and the maximum number of events/block for writes. It can also read various quantities including the total number of events in a file or buffer opened for reading or writing.

This routine can obtain a pointer to allocated memory containing the most recently read block header. The size of the memory is 14, 32-bit unsigned integers (words) 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.

To summarize, the ***request*** parameter can be the case independent string value of:

1) "B" for setting target block size in words
2) “W” for setting writing (to file) internal buffer size in words
3) "N" for setting max # of events/block
4) “R” for setting run number (used in file splitting)
5) “T” for setting run type (used in file splitting)
6) “S” for setting file split size in bytes
7) “M” for setting stream id (used in auto file naming)
8) "V" for getting evio version #
9) "H" for getting 14 words of block header info (only 8 valid for version < 6)
10) "E" for getting # of events in file/buffer

The ***argp*** parameter is a:

1) pointer to 32 bit unsigned int containing block size in 32-bit words if request = B
2) pointer to 32 bit unsigned int containing buffer size in 32-bit words if request = W
3) pointer to 32 bit unsigned int containing max # of events/block if request = N
4) pointer to 32 bit unsigned int containing run # if request = R
5) pointer to character containing run type if request = T
6) pointer to 64 bit unsigned int containing file split size in bytes if request = S
7) pointer to uint32_t containing stream id if request = M, or
8) pointer to 32 bit int returning the version # if request = V
9) address of pointer to unsigned 32 bit int returning a pointer to 14 uint32_t's of block header if request = H. (This pointer must be freed by caller since it points to allocated memory).
10) pointer to unsigned 32 bit int returning the total # of original events in existing file/buffer when reading or appending if request = E

3.5 *String manipulation*

In order to facilitate the handling of strings, 2 routines are provided. The first,

int evBufToStrings(char *buffer, int bufLen, char ***pStrArray,
int *strCount),

takes evio string format data and converts it into an array of strings. The second,

int evStringsToBuf(uint32_t *buffer, int bufLen, char **strings,
int stringCount, int *dataLen),

does the reverse and takes an array of strings and places them in evio format into a buffer.

3.6 *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.

3.7 *Dictionary*

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)

Note that if a file is being split, each file contains the dictionary. This routine is only implemented for evio version 4. Version 6 does not read or write dictionaries in order to keep the library simple. Use the C++ library if dictionaries are needed.

3.8 *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.

3.9 *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 "scons doc". Then simply view the doc/doxygen/C[or CC]/html/index.html file in a browser.

Chapter 4

4 C++ API Basics

4.1 *Basics*

There are things to know before reading and writing evio format files. To see the technical format details, go to chapter 11. However, this is **not** intended to be a full evio format tutorial. First, let's look at the classes which form the basis of evio.

Evio's container structures are called banks, segments, and tagsegments. These entities are implemented with 4 different classes. They all provide the same function of containing block data but differ in their header format. The very top level of an evio structure is always a bank and is called an event. This is represented by the ***EvioEvent*** class which is just a special case (subclass) of an ***EvioBank*** with a little extra private data included. Banks 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 access the information about an evio structure contained in its header, call getHeader() with event, bank, segment, or tagsegment objects. Using the returned ***BaseStructureHeader*** object, there are methods available to get & set values for its contents including type, tag, number, length, and padding.

Events of any complexity can be created using either the ***EventBuilder*** or ***CompactEventBuilder*** classes. The writing of events is done through ***EventWriter***, and the reading of events through ***EvioReader*** or ***EvioCompactReader***.

4.2 *Three interfaces*

The Java implementation of the evio library was developed first and the C++ was ported from that. Since Java has performance issues surrounding the creation and tracking of

objects, additional classes (i.e. a second Java API) was developed to minimize object creation and thereby improve performance. As a result, there are now 2 different APIs to use in handling evio data. Each will be described in different sections.

There is also a third interface (also originally written in Java and ported to C++) which treats events as buffers with any type of data, that is, it's data format agnostic. This API reads, writes, compresses and uncompresses data of any type. The other 2 API's call this one for all evio 6.0 operations. The reading of earlier formats is done with code from the previous version. This API will be described in its own section as well.

4.3 *Shared Pointers*

Evio data is a represented by a tree structure. Each evio structure, EvioEvent, EvioBank, EvioSegment, and EvioTagSegment, contains references to both parent and children. In order to facilitate implementation in a safe way, all these references are shared pointers.

4.4 *ByteBuffer Class*

The ***ByteBuffer*** class is a transplant from Java. It's not a complete port due entirely to the big differences between C++ and Java. However, it is extremely useful, saves a lot of effort, automagically takes care of endian issues, and it's used in much of the API. Here's a little tutorial, but the reader can always read the volumes written on the Java version of this class to get a complete handle on it. Remember, if something about the interface seems weird or inefficient, it's a translation from the Java.

4.4.1 Basic Usage

In C++, a ByteBuffer is an object which wraps an array of uint8_t type. This object provides a set of methods that makes it easier to work with the memory block. Using a ByteBuffer to write and read data typically follows this little 4-step process:

1. Write data into the buffer
2. Call buffer.flip()
3. Read data out of the buffer
4. Call buffer.clear() or buffer.compact()

When you write data into a buffer, the buffer keeps track of how much data you have written. Once you need to read the data, you can prepare the buffer for this by using the flip() method. The buffer lets you read all the data written into the buffer. Once done reading, you can clear the buffer, to make it ready for writing again by calling clear() or compact().

A buffer has four properties you should be familiar with. These are:

- **Capacity**: the maximum number of byte elements the buffer can hold. The capacity is set when the buffer is created and can never be changed.

- **Position**: the index of the next element to be read or written.

- **Limit**: the first element of the buffer that should not be read or written. In other words, the count of live elements in the buffer. The limit is less than or equal to the capacity.

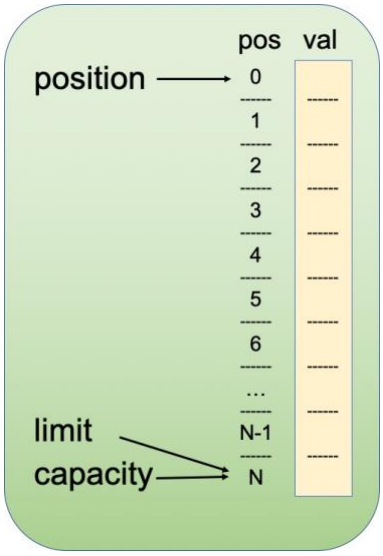
- **Mark**: a remembered position. Calling mark() sets mark = position. Calling reset( ) sets position = mark. The mark is undefined until set.

When you write data into the ByteBuffer, you do so at a certain position. Initially the position is 0. There are 2 types of writing: 1) an absolute write which does **not** advance its position after the write, and 2) a relative write which when a byte, long etc. has been written into the ByteBuffer, the position is advanced to point to the next position in the buffer to insert data into. Position can maximally become capacity - 1.

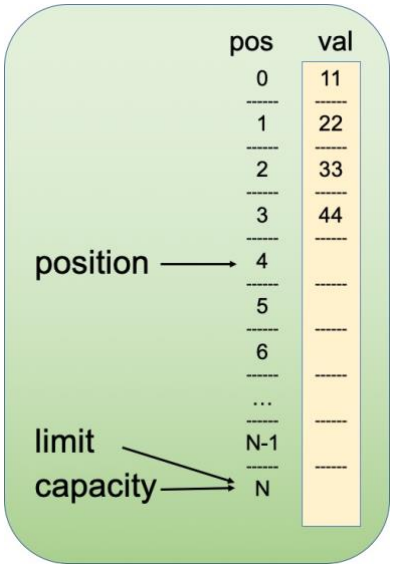
When you read data from a Buffer you also do so from a certain position. When you flip a ByteBuffer from writing to reading, it's limit is set the current position and the position is reset back to 0, thus preparing it to be read. There are 2 types of reading: 1) an absolute read which does **not** advance its position after the read and 2) a relative read which does. In the relative case, as you read data from the ByteBuffer the position is advanced to next position to read.

4.4.2 Example Diagrams

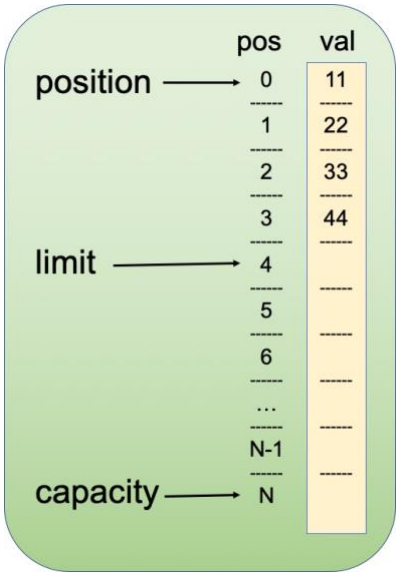
Initial buffer state in which position = 0, limit = capacity = N.
Also the state after calling clear().



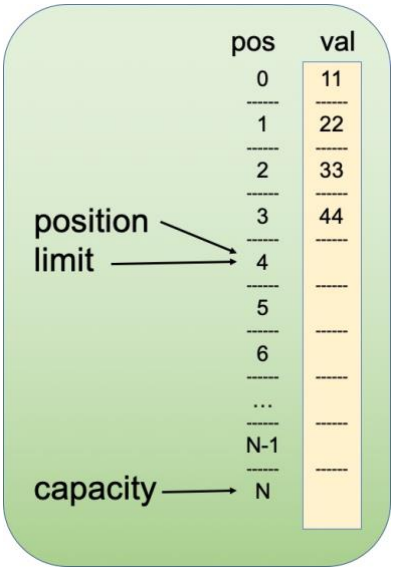
B**uffer after relative write of 4-byte integer = 0x11223344 (big endian)**



Buffer after flip()



Buffer after relative read of one 4-byte integer



4.4.3 Creating a Buffer

When creating a ByteBuffer object you can specify the amount memory in its buffer. If not specified it allocates 4096 bytes. Here is an example showing the allocation of a ByteBuffer, with a capacity of 128kBytes:

```
ByteBuffer buf(128000);
```

4.4.4 Writing Data

You can write data into a buffer via its put() methods. The put() method is overloaded and other variants of it exist, allowing the writing of different types of data or an array of bytes into the buffer. Here is an example showing how to do a relative write which places the written value at the buffer's current position and afterwards advances the position by the size of the write (4 bytes in this case):

```
int myInt = 123;
buffer.putInt(myInt);
```

Here's an example of writing at a specified position after which the position does not change:

```
size_t position = 1024;
int myInt = 123;
buffer.putInt(position, myInt);
```

4.4.5 The flip() method

The ***flip*** method prepares a ByteBuffer for reading after being written to. It sets the position back to 0, and sets the limit to where position just was. In other words, position now marks the reading position, and limit marks how many bytes, chars etc. were written into the buffer - the limit of how many bytes, chars etc. that can be read.

4.4.6 Reading Data

You can read data from the buffer using one of the get methods. They allow you to read data in different ways. Here is an example of a relative read at the current position which afterwards advances the postion by the size of the read (8 bytes in this case):

```
int64_t myLong = buf.getLong();
```

Here's an example of an absolute read at a specified position after which the position does not change:

```
size_t position = 1024;
double myDouble = buffer.getDouble(position);
```

4.4.7 Endianness

The transparent handling of the data's endianness (big or little) is a feature of ByteBuffers which makes reading and writing much easier. The endianness can be set by calling the order() method. This works with the ByteOrder class in the following way:

```
buf.order(ByteOrder::ENDIAN_BIG);
```

This method directs each write to place the data into the ByteBuffer's internal array in the prescribed endian. It also directs each read to retrieve that data properly. Thus, the endian value of a ByteBuffer can be set once and all reads and writes will transparently do the right thing. The default endian value of a ByteBuffer object is the endian value of the local node.

4.4.8 The rewind() method

The ***rewind()*** method sets the position back to 0, so you can reread all the data in the buffer. The limit remains unchanged, thus still marking how many bytes that can be read from the buffer.

4.4.9 The clear() and compact() methods

Once done reading data, one can make the buffer ready for writing again by calling either the ***clear()*** or ***compact()*** method. The clear() method does not erase any data, it merely sets the ByteBuffer back to its initial state (position = 0, limit = capacity).

The compact() method only clears the data which you have already read. If there is still unread data in the buffer that needs to be read later, but some writing needs to be done first, call compact() instead of clear(). This will copy all unread data to the beginning of the buffer. Then it sets position to right after the last unread element. The limit is set to capacity, just as in clear(). Now the buffer is ready for writing, but you will not overwrite the unread data.

4.4.10 The mark() and reset() methods

You can mark a given position in a buffer by calling the ***mark()*** method. You can then later reset the position back to the marked position by calling the ***reset()*** method.

4.4.11 The array() method

You can directly access the internal array that stores the data in a ByteBuffer by calling lthe ***array()*** method. For those readers familiar with it, all C++ ByteBuffers (unlike Java) are backed by arrays.

Chapter 5

5 Evio Specific C++ APIs

As mentioned in a previous section, there are 2 APIs which only deal with evio data. Where they differ is that one uses the classes ***EventBuilder*** and ***EvioReader*** to build events and read data, while the other uses ***CompactEventBuilder*** and ***EvioCompactReader***. So, what are the differences?

When communicating EvioEvents objects over the network, each object (and all of its contained objects in its tree structure) must be serialized into an array or buffer of bytes when sending and must deserialize the same bytes back into objects on the receiving end. This can lead to a serious performance penalty, especially in the original Java. To avoid having to serialize and deserialize continually, a new API was developed to allow the handling of evio data in byte buffer form. For lack of a better term, **compact** was the word chosen to describe it since all evio data handled in this API are contained in ByteBuffer objects and never expanded into EvioEvent objects.

5.1 *Regular API for Building Events*

The ***EventBuilder*** class constructs evio events. It takes care of all the little details and requires only the initial calling of a 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. It acts as a DOM-like model and so elements of the tree can be added anywhere at any time.

The following code uses the EventBuilder to create an event (almost always a bank of banks) with 3 children (a bank of banks, a bank of segments, and a bank of tagsegments). Each of these children have children of their own. It's written out to a buffer. The code relies on the ***EvioEvent, EvioBank, EvioSegment,*** and ***EvioTagSegment*** classes to get, fill, and update their internal data vectors:

```
// Data to write stored in these vectors
vector<uint8_t> byteVec;
vector<uint32_t> intVec;
vector<double> doubleVec;
vector<string> stringsVec;
```

```
//-------------------------------------
// Build event (bank of banks) with EventBuilder object
uint32_t tag = 1, num = 1;
EventBuilder builder(tag, DataType::BANK, num);
shared_ptr<EvioEvent> event = builder.getEvent();

//-------------------------------------
// First child of event = bank of banks
auto bankBanks = EvioBank::getInstance(tag+1, DataType::BANK, num+1);
// Add this bank as child of event
builder.addChild(event, bankBanks);

// Create first (& only) child of bank of banks = bank of ints
auto bankInts = EvioBank::getInstance(tag+11, DataType::UINT32, num+11);
// Get its internal vector of int data
auto &iData = bankInts->getUIntData();
// Write our data into that vector
iData.insert(iData.begin(), intVec.begin(), intVec.end());
// Done writing so tell builder to update its internals for this bank
bankInts->updateUIntData();
// Add this bank as child of bankBanks
builder.addChild(bankBanks, bankInts);

//-------------------------------------
// Second child of event = bank of segments
auto bankSegs = EvioBank::getInstance(tag+2, DataType::SEGMENT, num+2);
builder.addChild(event, bankSegs);

// Create first child of bank of segments = segment of doubles
auto segDoubles = EvioSegment::getInstance(tag+22, DataType::DOUBLE64);
auto &sdData = segDoubles->getDoubleData();
sdData.insert(sdData.begin(), doubleVec.begin(), doubleVec.end());
segDoubles->updateDoubleData();
builder.addChild(bankSegs, segDoubles);

// Create second child of bank of segments = segment of bytes
auto segBytes = EvioSegment::getInstance(tag+23, DataType::CHAR8);
auto &scData = segBytes->getCharData();
scData.insert(scData.begin(), byteVec.begin(), byteVec.end());
segBytes->updateCharData();
builder.addChild(bankSegs, segBytes);

//-------------------------------------
// Third child of event = bank of tagsegments
auto bankTsegs = EvioBank::getInstance(tag+3, DataType::TAGSEGMENT, num+3);
builder.addChild(event, bankTsegs);

// Create first child of bank of tagsegments = tagsegment of strings
auto tsegStrings = EvioTagSegment::getInstance(tag+33, DataType::CHARSTAR8);
auto &tstData = tsegStrings->getStringData();
tstData.insert(tstData.begin(), stringsVec.begin(), stringsVec.end());
tsegStrings->updateStringData();
builder.addChild(bankTsegs, tsegStrings);

//-------------------------------------
// Remove first segment (and all descendants) in bank of segments
builder.remove(segDoubles);

//-------------------------------------
// Take event, write it into buffer, get buffer ready to read
```

```
shared_ptr<ByteBuffer> buffer;
event->write(*(buffer.get()));
buffer->flip();
```

In addition to the methods for creating and adding banks, segments and tagsegments, there are methods to add all the various data types like chars, shorts, ints, longs, doubles, floats, and strings. The only tricky thing is handling Composite format data. Look at the relevant section to get more information on handling this complicated format.

Another way of creating events avoids the use of the EventBuilder class altogether. Call the insert() method of each structure to place it into another structure as a child. This way of doing things 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 abbreviated code does what the previous example does:

```
// Create an event
auto event = EvioEvent::getInstance(tag, DataType::BANK, nm);

//-------------------------------------
// First child of event = bank of banks
auto bankBanks = EvioBank::getInstance(tag+1, DataType::BANK, num+1);
event->insert(bankBanks, 0);

// Create first (& only) child of bank of banks = bank of ints
auto bankInts = EvioBank::getInstance(tag+11, DataType::UINT32, num+11);
auto &iData = bankInts->getUIntData();
iData.insert(iData.begin(), intVec.begin(), intVec.end());
bankInts->updateUIntData();
bankBanks->insert(bankInts, 0);

//-------------------------------------
// Second child of event = bank of segments
auto bankSegs = EvioBank::getInstance(tag2, DataType::SEGMENT, num+2);
event->insert(bankSegs, 1);

// Create first child of bank of segments = segment of bytes
auto segBytes = EvioSegment::getInstance(tag+22, DataType::CHAR8);
auto &scData = segBytes->getCharData();
scData.insert(scData.begin(), byteVec.begin(), byteVec.end());
segBytes->updateCharData();
bankSegs->insert(segBytes, 0);

//-------------------------------------
// Third child of event = bank of tagsegments
auto bankTsegs = EvioBank::getInstance(tag+3, DataType::TAGSEGMENT, num+3);
event->insert(bankTsegs, 2);

// Create first child of bank of tagsegments = segment of ints
auto tsegInts = EvioTagSegment::getInstance(tag+33, DataType::UINT32);
auto &tiData = tsegInts->getUIntData();
tiData.insert(tiData.begin(), intVec.begin(), intVec.end());
tsegInts->updateUIntData();
bankTsegs->insert(tsegInts, 0);

//-------------------------------------
// If doing things this way, be sure to set all the header lengths
```

```
event->setAllHeaderLengths();
```

5.2 *Compact API for Building Events*

The ***CompactEventBuilder*** class also constructs evio events. It takes care of all the little details and requires only the initial calling of a constructor to which a buffer is provided. It acts as a SAX-like model and so each structure is added depth-first to the tree.

The following code uses the CompactEventBuilder to create an event (as above) with 3 children (a bank of banks, a bank of segments, and a bank of tagsegments). Each of these children have children of their own. The builder writes it into a buffer:

```
// Data to write stored in these arrays
int dataElementCount = 10;
uint8_t byteArray[];
uint32_t intArray[];
double doubleArray[];
vector<string> stringsVec;

// Create the buffer to hold everything
auto buffer = make_shared<ByteBuffer>(8192);

// Build event (bank of banks) with CompactEventBuilder object
uint32_t tag = 1, num = 1;
CompactEventBuilder builder(buffer);

//-------------------------------------
// add top/event level bank of banks
builder.openBank(tag, DataType::BANK, num);


// add bank of banks to event
builder.openBank(tag + 1, DataType::BANK, num + 1);

// add bank of ints to bank of banks
builder.openBank(tag + 11, DataType::UINT32, num + 11);
builder.addIntData(intArray, dataElementCount);
// done with bank of ints, go to enclosing structure
builder.closeStructure();

// done with bank of banks
builder.closeStructure();


//-------------------------------------
// add bank of segs to event
builder.openBank(tag + 2, DataType::SEGMENT, num + 2);

// add first child, tagseg of doubles to bank of segs
builder.openTagSegment(tag + 22, DataType::DOUBLE64);
builder.addDoubleData(doubleArray, dataElementCount);
builder.closeStructure();

// add second child, seg of bytes to bank of segs
builder.openSegment(tag + 23, DataType::CHAR8);
builder.addByteData(byteArray, dataElementCount);
```

```
    builder.closeStructure();

    // done with bank of segs
    builder.closeStructure();


    //-------------------------------------
    // add bank of tagsegs
    builder.openBank(tag + 3, DataType::TAGSEGMENT, num + 3);

    // add tagseg of strings
    builder.openTagSegment(tag + 21, DataType::CHARSTAR8);
    builder.addStringData(stringsVec);
    builder.closeStructure();

    // Close all existing structures
    builder.closeAll();

    // There is no way to remove any structures

    // Get ready-to-read buffer from builder
    // (this sets the proper pos and lim in buffer)
    auto bytebbuffer = builder.getBuffer();
```

Notice in particular the last line above in which the buffer (given as an argument to the builder's constructor) is obtained through calling getBuffer(). This method returns that same buffer with its position and limit properly set for reading.

5.3 *Tree structure API for Building Events*

In addition to using the "builder" classes for creating events, one can also use the methods used to handle the tree structure of the EvioEvent, EvioBank, EvioSegment, and EvioTagsegment classes to create events. The methods used for handling the tree structure are extensive and are, in fact, translated from Java's DefaultMutableTreeNode class. To get a more complete picture, see the doxygen documentation in which each method is thoroughly described. The only tricky thing is to remember to call a structure's updateXXXData() method after data has been modified in order to reflect those changes. To get a sampling, look at the following example:

```
// Create some banks
auto topBank   = EvioBank::getInstance(0, DataType::BANK, 0);
auto midBank   = EvioBank::getInstance(1, DataType::BANK, 1);
auto midBank2  = EvioBank::getInstance(2, DataType::SEGMENT, 2);
auto childBank = EvioBank::getInstance(4, DataType::FLOAT32, 4);

// Child bank's float data
auto &fData = childBank->getFloatData();
fData.push_back(0.);
fData.push_back(1.);
childBank->updateFloatData();

// Create tree
topBank->add(midBank);
topBank->add(midBank2);
midBank->add(childBank);
```

```
// Create more structures
auto childSeg1 = EvioSegment::getInstance(5, DataType::INT32);
auto childSeg2 = EvioSegment::getInstance(6, DataType::INT32);
auto childSeg3 = EvioSegment::getInstance(7, DataType::SHORT16);

// Children data
auto &iData = childSeg1->getIntData();
iData.push_back(3);
iData.push_back(4);
childSeg1->updateIntData();

auto &iData2 = childSeg2->getIntData();
iData2.push_back(5);
iData2.push_back(6);
childSeg2->updateIntData();

auto &sData = childSeg3->getShortData();
sData.push_back(7);
sData.push_back(8);
childSeg3->updateShortData();

// Add segments to tree
midBank2->add(childSeg1);
midBank2->add(childSeg2);
midBank2->add(childSeg3);

//--------------------
// Print out tree info
//--------------------
cout << std::boolalpha;
cout << "Is child descendant of Top? " << topBank->isNodeDescendant(childBank) << endl;
cout << "Is Top ancestor of child? " << childBank->isNodeAncestor(topBank) << endl;

cout << "Depth at Mid bank = " << midBank->getDepth() << endl;
cout << "Depth at Child bank = " << childBank->getDepth() << endl;
cout << "Level at top bank = " << topBank->getLevel() << endl;
cout << "Level at child = " << childBank->getLevel() << endl;

cout << "midBank2 has " << midBank2->getChildCount() << " children" << endl;
cout << "Remove childSeg2 from midBank2:" << endl;
midBank2->remove(childSeg2);
cout << "midBank2 now has " << midBank2->getChildCount() << " children" << endl;

cout << "CALL sharedAncestor for both mid banks" << endl;
auto strc = midBank2->getSharedAncestor(midBank);
if (strc != nullptr) {
    cout << "shared ancestor of midBank 1&2 = " << strc->toString() << endl;
}
else {
    cout << "There is NO shared ancestor of midBank 1&2" << end;
}

auto path = childBank->getPath();
cout << "Path of childBank:" << endl;
for (auto str : path) {
    cout << "     -  " << str->toString() << endl;
}

cout << "iterate thru topBank children" << endl;
auto beginIter = topBank->childrenBegin();
auto endIter   = topBank->childrenEnd();
```

```
for (; beginIter != endIter; beginIter++) {
    auto kid = *beginIter;
    std::cout << "  kid = " << kid->toString() << std::endl;
}


auto root = childBank->getRoot();
cout << "Root of childBank is = " << root->toString() << endl;

cout << "Is childBank root structure? " << childBank->isRoot() << endl;
cout << "Is topBank root structure? " << topBank->isRoot() << endl;

shared_ptr<BaseStructure> node = topBank;
cout << "Starting from root:" << endl;
do {
    node = node->getNextNode();
    if (node == nullptr) {
        cout << "  next node = nullptr" << endl;
    }
    else {
        cout << "  next node = " << node->toString() << endl;
    }
} while (node != nullptr);


cout << "childSeg1 is a Leaf? " << childSeg1->isLeaf() << endl;
cout << "midBank2 leaf count = " << midBank2->getLeafCount() << endl;
```

5.4 *Writing Events*

5.4.1 Writing to file or buffer

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 which allow the user to chose whether to append to or overwrite any previously existing data. The user can also set the maximum record size and the maximum number of events per record as well as specify a dictionary and data byte order among other things. Refer to the doxygen documentation for all of the possibilities.

Below is 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 the ***ByteBuffer*** class, take some time to read up on it when using buffers. It will allow you to do many things.

```
// Append or overwrite
bool append = false;
// File's name
string filename = "./myData";
// Byte order in which to write data
ByteOrder order = ByteOrder::ENDIAN_LOCAL;

// Create an EventWriter object to write out events to file
EventWriter writer(filename, order, append);

// Create an event (see previous section)
EventBuilder builder(1, DataType::BANK, 1);
```

```
// Add structures & data here …

// Get reference to the event just created
auto event = builder.getEvent();

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

// Finish the writing
writer.close();
```

And now the same example for buffers:

```
// Create a ByteBuffer and get a shared pointer to it
auto myBuf = std::make_shared<ByteBuffer>(1024);

// Byte order in which to write data
myBuf.order(ByteOrder::ENDIAN_LITTLE);

// Create an EventWriter object to write out events to buffer
EventWriter writer(myBuf);

// Create an event (see previous section)
EventBuilder builder(1, DataType::BANK, 1);

// Add structures & data here …

// Get reference to the event just created
auto event = builder.getEvent();

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

// Finish the writing
writer.close();
```

5.4.2 Naming files

The filename passed to any of the constructors may contain characters of the form **$(env)** where “env” is the name of an environmental variable. When the file is created, all such constructs will be substituted with the actual environmental variable’s value (or nothing if it doesn’t exist).

Similarly, the filename may contain constructs of the form **%s** which will be substituted with the actual run type’s value (if passed in as a parameter to the constructor).

The filename may also contain the run number value (if passed in as a parameter to the constructor) and the split number (if splitting). This is done by allowing up to 2, C-style integer format specifiers (such as %03d, or %x) in the filename. If more than 2 are found, an exception will be thrown. If no "0" precedes any integer between the "%" and the "d" or "x" of the format specifier, it will be added automatically in order to avoid spaces in the generated filename. The first occurrence will be substituted with the given run number value. If the file is being split, the second will be substituted with the split number. If 2 specifiers exist and the file is not being split, no substitutions are made.

5.4.3 Splitting files

When writing significant amounts of data to a single file, that file can get very large – too large. Historically, run control was able to split the data into multiple files with an automatic file naming system. The ability to split files is now built into evio as is the naming system. Simply pick the constructor designed for file splitting with parameters allowing the user to choose the number of bytes at which to start writing to a new file and the name of the files to use. The constructor of both the EventWriter and EvioCompactEventWriter (more on this in the next chapter) classes have input parameters for a base filename, run type, run number, and split size.

A description of the general file naming system is in the section above, but when splitting into multiple files (split size > 0), the user should also be aware that the generated files names are distinguished by a split number. If the base filename contains C-style int format specifiers, then the first occurrence will be substituted with the given run number value. The second will be substituted with the split number. If no specifier for the split number exists, it is automatically tacked onto the end of the file name.

Below is example code with comments showing how the file naming and splitting is done.

```
int split           = 100000000; // split at 100MB
int runNumber       = 1;
string runType      = "myExperiment";
string directory    = "/myDirectory";
string baseFilename = "my$(BASE_NAME)_%s_%x_%03d.ext";
EventWriter writer(baseFilename, directory,
                   runType, runNumber, split,
                   64000, 1000, 300000,
                   byteOrder, dictionary,
                   bitInfo, overWriteOK, append);
```

The baseFilename string will have the value of the environmental variable BASE_NAME, substituted for the $(BASE_NAME) part of the string along with the runType substituted for the %s, the runNumber substituted for the %x (hex format), and the split number substituted for the %03d. If BASE_NAME has the value "File", then the first 3 split files will have the names:

```
myFile_myExperiment_1_001.ext
myFile_myExperiment_1_002.ext
myFile_myExperiment_1_003.ext
```

5.5 *Regular API for Reading Events*

Start reading an evio format file or buffer with an ***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 record numbers are sequential. (Find out about record numbers by reading through the section which describes the evio

file format). There is also an optional parameter for choosing between sequential and random-access methods for reading a file.

There are two basic types of read calls. The first is random-access, and the second is sequentially in nature. The following are the random-access methods of the EvioReader class:

```
getEvent(size_t i)
parseEvent(size_t i)
gotoEventNumber(size_t i)
```

and the sequential methods:

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

When mixing calls of these two categories in one application, be aware that calling a random-access method will affect the sequential methods. The sequence will jump to the event called by event number. For example, if an application calls parseNextEvent() twice, then suddenly calls parseEvent(20), the next time parseNextEvent() is called, it will returned the 21st event, not the 3rd.

Now for a word on performance. 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.

```
// READ a file
string filename = "myFile";
EvioReader reader(filename);

// Find out how many events the file contains
int32_t evCount = reader.getEventCount();
cout << "File has " << evCount << " events" << endl;

// Does it contain a dictionary?
if (reader.hasDictionary()) {
    string dict = reader.getDictionaryXML();
    cout << "Dictionary = " << dict << endl;
}

// Does it contain a first event?
shared_ptr<EvioEvent> fe = reader.getFirstEvent();
if (fe != nullptr) {
    cout << "First event size = " << fe->getTotalBytes() << " bytes" << endl;
```

```
}

// Look at each event with random-access type method
cout << "Print out regular events' raw data:" << endl;
for (int i = 0; i < evCount; i++) {
    auto ev = reader.parseEvent(i + 1);

    auto & dataVec = ev->getRawBytes();
    Util::printBytes(dataVec.data(), dataVec.size(),
                     "  Event #" + to_string(i));
}

// Go back to beginning of file
reader.rewind();

// Use sequential access to events
shared_ptr<EvioEvent> ev = nullptr;
while ((ev = evioReader.parseNextEvent()) != nullptr) {
    // do something with event
    cout << "Event has tag = " << ev->getHeader()->getTag() << endl;
}
```

5.6 *Searching*

Most users are interested in searching for a specific event or an evio structure. To this end, evio has several built-in searches for ease of use. See the doxygen documentation for the static methods in 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 vector. In addition, listeners can be added to execute methods upon the finding of a matching structure using ***IEvioListener*** interface.

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.

```
// Search for banks with specific tag/num using StructureFinder
cout << "Search for banks of tag=4 & num=4 using StructureFinder, got:" << endl;

uint16_t tag = 4;
uint8_t  num = 4;
std::vector<std::shared_ptr<BaseStructure>> vec;

// Searching the topBank structure
StructureFinder::getMatchingBanks(topBank, tag, num, vec);
for (auto n : vec) {
    std::cout << "  bank = " << n->toString() << std::endl;
}
vec.clear();

// Custom search for segments with odd tags
cout << "Search for segments with odd numbered tags, got:" << endl;

// Define custom filter
class myFilter : public IEvioFilter {
    public:
```

```
        bool accept(StructureType const &t, shared_ptr<BaseStructure> s) override){
             return ((t == StructureType::STRUCT_SEGMENT) &&
                     (s->getHeader()->getTag() % 2 == 1));
        }
};

// Use filter in method which does matching
auto filter = make_shared<myFilter>();
topBank->getMatchingStructures(filter, vec);
for (auto n : vec) {
    cout << "  bank = " << n->toString() << endl;
}

// Custom search for segments with odd tags which also executes listener
cout << "Search for all segs with odd tags, & execute listener:" << endl;

// Define custom listener
class myListener : public IEvioListener {
    public:
        void gotStructure(std::shared_ptr<BaseStructure> topStructure,
                          std::shared_ptr<BaseStructure> structure) {
        cout << "  listener run for structure = " << structure->toString() << endl;
    }

    // We're not parsing, so these are not used ...
    void startEventParse(std::shared_ptr<BaseStructure> structure) {}
    void endEventParse(std::shared_ptr<BaseStructure> structure) {}
};

// Do the searching
auto listener = make_shared<myListener>();
topBank->visitAllStructures(listener, filter);
```

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

5.7 *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.

Evio 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:

1. after a structure is read in and filter applied while an event is being searched/parsed
2. just before an event (bank, segment, or tagsegment) is to be parsed, and
3. just after an event has been parsed.

Here again, the previous section has a simple example of doing this. Following is an example of code that uses both a listener and a filter, but this time in the context of parsing.

```
// Read some data
EvioReader reader(buffer);
auto parser = reader.getParser();

// Define a listener for the parser
class myListener : public IEvioListener {

  public:
    // Call after structure is read in & filter applied while
    // event is being parsed
    void gotStructure(shared_ptr<BaseStructure> topStructure,
                      shared_ptr<BaseStructure> structure) override {
        cout << "GOT struct = " << structure->toString() << endl;
    }

    // Call at start of event parsing
    void startEventParse(shared_ptr<BaseStructure> structure) override {
        cout << "START parsing event = " << structure->toString() << endl;
    }

    // Called at end of event parsing
    void endEventParse(shared_ptr<BaseStructure> structure) override {
        cout << "END parsing event = " << structure->toString() << endl;
    }
};

auto listener = make_shared<myListener>();

// Add the listener to the parser
parser->addEvioListener(listener);

// Define a filter to select everything (not much of a filter!)
class myFilter : public IEvioFilter {
   public:
    bool accept(StructureType const & t, std::shared_ptr<BaseStructure> s) override {
        return (true);
    }
};

auto filter = std::make_shared<myFilter>();

// Add the filter to the parser
parser->setEvioFilter(filter);

// Now parse an event
cout << "Run custom filter & listener on first event:" << endl;
auto ev = reader.parseEvent(1);
```

5.8 *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 and the differences in the format of their header words. For example:

```
// Take an existing EvioSegment
shared_ptr<EvioSegment> seg;

// Turn that segment into a bank with num = 10
uint8_t num = 10;
shared_ptr<EvioBank> bank = StructureTransformer::transform(seg, num);
```

5.9 *Dictionaries*

This section describes how dictionaries can be used (refer to relevant section for more details). Define a dictionary by something like:

```
// Define xml dictionary String
string xmlDictString =
      "<xmlDict>\n" +
      "  <dictEntry name=\"myEntry\"   tag=\"1\"  num=\"1\" />\n" +
      "  <dictEntry name=\"second bank\"  tag=\"2\"  num=\"2\" />\n" +
      "  <bank name=\"parentEntry\"  tag=\"10\"  num=\"3\" >" +
      "    <leaf name=\"childEntry\"  tag=\"20\"  num=\"4\" />" +
      "   </bank>" +
      "</xmlDict>";

// Create a dictionary object from xml string
EvioXmlDictionary dict(xmlDictString);
```

Once the dictionary is created, 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. There three methods in this class that use the dictionary as seen below:

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

// Names to look for
string myName     = "myEntry";
string childName  = "childEntry";
string parentName = "parentEntry";

// Place to store search results
vector<shared_ptr<BaseStructure>> vec;

// Search an event for a structure which matches an entry
// in this dictionary with a particular name
StructureFinder::getMatchingStructures(event, myName, dict, vec);

// Search for structures whose parent has a particular name
StructureFinder::getMatchingParent(event, parentName, dict, vec);

// Search for structures who have a child with a particular name
StructureFinder::getMatchingChild(event, childName, dict, vec);

// Print out the list of structures
```

```
for (auto & struc : vec) {
    cout << "found struct -> " << struc->toString() << endl;
}
```

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's getDictionaryXML()*** and ***hasDictionaryXML()*** methods. Similarly, the ***EvioCompactReader*** has ***getDictionaryXML()***, ***getDictionary()***, and ***hasDictionary()*** methods. 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 if 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
 stringstream ss;

 ss << "<xmlDict>\n" <<
 " <dictEntry name=\"me\" tag=\"10\" num=\"0\" type=\"COMPOSITE\" />\n" <<
 "   <description format=\"2iN(FD)\" >\n" <<
 "        Any comments can go right here!" <<
 "   </description>\n" <<
 "  </dictEntry>\n" <<
 "</xmlDict>";

string xmlDictString = ss.str();

// Create a dictionary object from xml string
EvioXmlDictionary dict(xmlDictString, 0);

// Retrieve & print info from dictionary
cout << "Getting stuff for name = \"me\":" << endl;
uint16_t tag;
dict.getTag("me", &tag);
cout << "    tag         = " << tag << endl;
uint8_t num;
dict.getNum("me",  &num);
cout << "    num         = " << +num << endl;
cout << "    type        = " << dict.getType("me") << endl;
cout << "    format      = " << dict.getFormat("me") << endl;
cout << "    description = " << dict.getDescription("me") << endl;

cout << "Getting stuff for tag = 10, num = 0:" << endl;
cout << "    type        = " << dict.getType(10,0) << endl;
cout << "    name        = " << dict.getName(10,0) << endl;
cout << "    format      = " << dict.getFormat(10,0) << endl;
cout << "    description = " << dict.getDescription(10,0) << endl;
```

One way to see what a dictionary contains using the ***dict.getMap()*** method.

```
cout << "Print out contents of dictionary:" << endl;
auto & map = dict.getMap();
for (auto & entry : map) {
    cout << "   " << entry.first << " :   " << entry.second->toString() << endl;
}
```

5.10 *First Event*

If the user wants the same (first) event in each split file, then simply select the ***EventWriter*** constructor that has an argument for the first event (for either file or buffer writing). An alternative method is to call the EventWriter's ***setFirstEvent()*** method. If calling the method, make sure it's called before any other events are written in order to ensure that it is written to each of the split files.

5.11 *XML format events*

If the user wants to view an event in xml format, use the Java library.

Chapter 6

6 Evio Specific Java APIs

As mentioned in a previous section, there are 2 APIs which only deal with evio data. Where they differ is that one uses the classes ***EventBuilder*** and ***EvioReader*** to build events and read data, while the other uses ***CompactEventBuilder*** and ***EvioCompactReader***. So, what are the differences?

When communicating EvioEvents objects over the network, each object (and all of its contained objects in its tree structure) must be serialized into an array or buffer of bytes when sending and must deserialize the same bytes back into objects on the receiving end. This can lead to a serious performance penalty, especially in the original Java. To avoid having to serialize and deserialize continually, a new API was developed to allow the handling of evio data in byte buffer form. For lack of a better term, **compact** was the word chosen to describe it since all evio data handled in this API are contained in ByteBuffer objects and never expanded into EvioEvent objects.

Note that this chapter follows closely after the pattern of the previous chapter on the C++ API. Not surprising since the C++ code was translated from the Java.

6.1 *Regular API for Building Events*

The ***EventBuilder*** class constructs evio events. It takes care of all the little details and requires only the initial calling of a 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. It acts as a DOM-like model and so elements of the tree can be added anywhere at any time.

The following code uses the EventBuilder to create an event (almost always a bank of banks) with 3 children (a bank of banks, a bank of segments, and a bank of tagsegments). Each of these children have children of their own. It's written

out to a buffer. The code relies on the ***EvioEvent, EvioBank, EvioSegment,*** and ***EvioTagSegment*** classes to get, fill, and update their internal data vectors:

```
// Data to write stored in these arrays
int[]    int1;
byte[]   byte1;
double[] double1;
String[] string1;

//-------------------------------------
// Build event (bank of banks) with EventBuilder object
int tag = 1, num = 1;
EventBuilder builder = new EventBuilder(tag, DataType.BANK, num);
event = builder.getEvent();

//-------------------------------------
// First child of event = bank of banks
EvioBank bankBanks = new EvioBank(tag+1, DataType.BANK, num+1);
builder.addChild(event, bankBanks);

// Create first (& only) child of bank of banks = bank of int
EvioBank bankInts = new EvioBank(tag+11, DataType.UINT32, num+11);
// Add data to bank
bankInts.appendIntData(int1);
// Add this bank as child of bankBanks
builder.addChild(bankBanks, bankInts);

//-------------------------------------
// Second child of event = bank of segments
EvioBank bankSegs = new EvioBank(tag+14, DataType.SEGMENT, num+14);
builder.addChild(event, bankSegs);

// Create first child of bank of segments = segment of doubles
EvioSegment segDoubles = new EvioSegment(tag+12, DataType.DOUBLE64);
segDoubles.appendDoubleData(double1);
builder.addChild(bankSegs, segDoubles);

// Create second child of bank of segments = segment of bytes
EvioSegment segBytes = new EvioSegment(tag+9, DataType.CHAR8);
segBytes.appendByteData(byte1);
builder.addChild(bankSegs, segBytes);

//-------------------------------------
// Third child of event = bank of tagsegments
EvioBank bankTsegs = new EvioBank(tag+15, DataType.TAGSEGMENT, num+15);
builder.addChild(event, bankTsegs);

// Create first child of bank of tagsegments = tagsegment of strings
EvioTagSegment tsegStrings = new EvioTagSegment(tag+21, DataType.CHARSTAR8);
tsegStrings.appendStringData(string1);
builder.addChild(bankTsegs, tsegStrings);

//-------------------------------------
// Remove first segment (and all descendants) in bank of segments
builder.remove(segDoubles);

//-------------------------------------
// Take event, write it into buffer, get buffer ready to read
event.write(buffer);
buffer.flip();
```

In addition to the methods for creating and adding banks, segments and tagsegments, there are methods to add all the various data types like chars, shorts, ints, longs, doubles, floats, and strings. The only tricky thing is handling Composite format data. Look at the relevant section to get more information on handling this complicated format.

Another way of creating events avoids the use of the EventBuilder class altogether. Call the insert() method of each structure to place it into another structure as a child. This way of doing things 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 abbreviated code does what the previous example does:

```
// Data to write stored in these arrays
int[]    int1;
byte[]   byte1;
double[] double1;
String[] string1;

//-------------------------------------
// Build event (bank of banks)
int tag = 1, num = 1;
EvioEvent event = new EvioEvent(tag, DataType.BANK, num);

//-------------------------------------
// First child of event = bank of banks
EvioBank bankBanks = new EvioBank(tag+1, DataType.BANK, num+1);
event.insert(bankBanks);

// Create first (& only) child of bank of banks = bank of int
EvioBank bankInts = new EvioBank(tag+11, DataType.UINT32, num+11);
bankInts.appendIntData(int1);
// Add this bank as child of bankBanks
bankBanks.insert(bankInts);

//-------------------------------------
// Second child of event = bank of segments
EvioBank bankSegs = new EvioBank(tag+14, DataType.SEGMENT, num+14);
event.insert(bankSegs);

// Create child of bank of segments = segment of bytes
EvioSegment segBytes = new EvioSegment(tag+9, DataType.CHAR8);
segBytes.appendByteData(byte1);
bankSegs.insert(segBytes);

//-------------------------------------
// Third child of event = bank of tagsegments
EvioBank bankTsegs = new EvioBank(tag+15, DataType.TAGSEGMENT, num+15);
event.insert(bankTsegs);

// Create first child of bank of tagsegments = tagsegment of strings
EvioTagSegment tsegStrings = new EvioTagSegment(tag+21, DataType.CHARSTAR8);
tsegStrings.appendStringData(string1);
bankTsegs.insert(tsegStrings);

//-------------------------------------
// If doing things this way, be sure to set all the header lengths
event.setAllHeaderLengths();
```

6.2 *Compact API for Building Events*

The ***CompactEventBuilder*** class also constructs evio events. It takes care of all the little details and requires only the initial calling of a constructor to which a buffer is provided. It acts as a SAX-like model and so each structure is added depth-first to the tree.

The following code uses the CompactEventBuilder to create an event (as above) with 3 children (a bank of banks, a bank of segments, and a bank of tagsegments). Each of these children have children of their own. The builder writes it into a buffer:

```
// Data to write stored in these arrays
byte[] byteArray;
int[]  intArray;
double[] doubleArray;
String[] stringsArray;

// Create the buffer to hold everything
ByteBuffer buffer = ByteBuffer.allocate(8192);

// Build event (bank of banks) with CompactEventBuilder object
int tag = 1, num = 1;
CompactEventBuilder builder = new CompactEventBuilder(buffer);

//-------------------------------------
// add top/event level bank of banks
builder.openBank(tag, num, DataType.BANK);


// add bank of banks to event
builder.openBank(tag+1, num+1, DataType.BANK);

// add bank of ints to bank of banks
builder.openBank(tag+11, num+11, DataType.UINT32);
builder.addIntData(intArray);
// done with bank of ints, go to enclosing structure
builder.closeStructure();

// done with bank of banks
builder.closeStructure();


//-------------------------------------
// add bank of segs to event
builder.openBank(tag+2, num+2, DataType.SEGMENT);

// add first child, tagseg of doubles to bank of segs
builder.openTagSegment(tag + 22, DataType.DOUBLE64);
builder.addDoubleData(doubleArray);
builder.closeStructure();

// add second child, seg of bytes to bank of segs
builder.openSegment(tag + 23, DataType.CHAR8);
builder.addByteData(byteArray);
```

```
builder.closeStructure();

// done with bank of segs
builder.closeStructure();


//-------------------------------------
// add bank of tagsegs
builder.openBank(tag+3, num+3, DataType.TAGSEGMENT);

// add tagseg of strings
builder.openTagSegment(tag + 21, DataType.CHARSTAR8);
builder.addStringData(stringsArray);
builder.closeStructure();

// Close all existing structures
builder.closeAll();

// There is no way to remove any structures

// Get ready-to-read buffer from builder
// (this sets the proper pos and lim in buffer)
ByteBuffer bytebbuffer = builder.getBuffer();
```

Notice in particular the last line above in which the buffer (given as an argument to the builder's constructor) is obtained through calling getBuffer(). This method returns that same buffer with its position and limit properly set for reading.

6.3 *Tree structure API for Building Events*

In addition to using the "builder" classes for creating events, one can also use the methods used to handle the tree structure of the EvioEvent, EvioBank, EvioSegment, and EvioTagsegment classes to create events. The methods used for handling the tree structure implement Java's MutableTreeNode interface. To get a more complete picture, see the doxygen documentation in which each method is thoroughly described. The only tricky thing is to remember to call a structure's updateXXXData() method after data has been modified in order to reflect those changes. To get a sampling, look at the following example:

```
// Create some banks
EvioBank topBank   = new EvioBank(0, DataType.BANK, 0);
EvioBank midBank   = new EvioBank(1, DataType.BANK, 1);
EvioBank midBank2  = new EvioBank(2, DataType.SEGMENT, 2);
EvioBank childBank = new EvioBank(4, DataType.FLOAT32, 4);

// Child bank's float data
float[] fData = new float[] {0.F, 1.F};
childBank.setFloatData(fData);

// Create tree
topBank.insert(midBank);
topBank.insert(midBank2);
midBank.insert(childBank);

// Create more structures
```

```
EvioSegment childSeg1 = new EvioSegment(5, DataType.INT32);
EvioSegment childSeg2 = new EvioSegment(6, DataType.INT32);
EvioSegment childSeg3 = new EvioSegment(7, DataType.SHORT16);

// Children data
int[] iData = new int[] {3, 4};
childSeg1.setIntData(iData);

int[] iData2 = new int[] {5, 6};
childSeg2.setIntData(iData2);

short[] sData = new short[] {7, 8};
childSeg3.setShortData(sData);

// Add segments to tree
midBank2.insert(childSeg1);
midBank2.insert(childSeg2);
midBank2.insert(childSeg3);

//--------------------
// Print out tree info
//--------------------
System.out.println("midBank2 has " + midBank2.getChildCount() + " kids");
System.out.println("Remove childSeg2 from midBank2");
midBank2.remove(childSeg2);
System.out.println("midBank2 now has " + midBank2.getChildCount() + "
kids");

System.out.println("iterate thru topBank's children:");
for (BaseStructure kid : topBank.getChildrenList()) {
    System.out.println("  kid = " + kid.toString());
}

System.out.println("childSeg1 is a Leaf? " + childSeg1.isLeaf());
```

6.4 *Writing Events*

6.4.1 Writing to file or buffer

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 which allow the user to chose whether to append to or overwrite any previously existing data. The user can also set the maximum record size and the maximum number of events per record as well as specify a dictionary and data byte order among other things. Refer to the doxygen documentation for all of the possibilities.

Below is 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.

```
// Append or overwrite
boolean append = false;
// File's name
String filename = "./myData";
// Byte order in which to write data
```

```
ByteOrder order = ByteOrder.BIG_ENDIAN;

// Create an EventWriter object to write out events to file
EventWriter writer = new EventWriter(filename, append, order);

// Create an event (see previous section)
EventBuilder builder = new EventBuilder(1, DataType.BANK, 1);

// Add structures & data here …

// Get reference to the event just created
EvioEvent event = builder.getEvent();

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

// Finish the writing
writer.close();
```

And now the same example for buffers:

```
// Create a ByteBuffer and get a shared pointer to it
ByteBuffer myBuf = ByteBuffer.allocate(1024);

// Byte order in which to write data
myBuf.order(ByteOrder.LITTLE_ENDIAN);

// Create an EventWriter object to write out events to buffer
EventWriter writer = new EventWriter(myBuf);

// Create an event (see previous section)
EventBuilder builder = new EventBuilder(1, DataType.BANK, 1);

// Add structures & data here …

// Get reference to the event just created
EvioEvent event = builder.getEvent();

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

// Finish the writing
writer.close();
```

6.4.2 Naming files

The filename passed to any of the constructors may contain characters of the form **$(env)** where “env” is the name of an environmental variable. When the file is created, all such constructs will be substituted with the actual environmental variable’s value (or nothing if it doesn’t exist).

Similarly, the filename may contain constructs of the form **%s** which will be substituted with the actual run type’s value (if passed in as a parameter to the constructor).

The filename may also contain the run number value (if passed in as a parameter to the constructor) and the split number (if splitting). This is done by allowing up to 2, C-style integer format specifiers (such as %03d, or %x) in the filename. If more than 2 are found, an exception will be thrown. If no "0" precedes any integer between the "%" and the "d" or "x" of the format specifier, it will be added automatically in order to avoid spaces in the generated filename. The first occurrence will be substituted with the given run number value. If the file is being split, the second will be substituted with the split number. If 2 specifiers exist and the file is not being split, no substitutions are made.

6.4.3 Splitting files

When writing significant amounts of data to a single file, that file can get very large – too large. Historically, run control was able to split the data into multiple files with an automatic file naming system. The ability to split files is now built into evio as is the naming system. Simply pick the constructor designed for file splitting with parameters allowing the user to choose the number of bytes at which to start writing to a new file and the name of the files to use. The constructor of both the EventWriter and EvioCompactEventWriter (more on this in the next chapter) classes have input parameters for a base filename, run type, run number, and split size.

A description of the general file naming system is in the section above, but when splitting into multiple files (split size > 0), the user should also be aware that the generated files names are distinguished by a split number. If the base filename contains C-style int format specifiers, then the first occurrence will be substituted with the given run number value. The second will be substituted with the split number. If no specifier for the split number exists, it is automatically tacked onto the end of the file name.

Below is example code with comments showing how the file naming and splitting is done.

```
int split           = 100000000; // split at 100MB
int runNumber       = 1;
String runType      = "myExperiment";
String directory    = "/myDirectory";
String baseFilename = "my$(BASE_NAME)_%s_%x_%03d.ext";
.
.
.
EventWriter writer  = new EventWriter(baseFileName, directory, runType,
                                      runNumber, split,
                                      maxRecordSize, maxEventCount,
                                      byteOrder, dictionary,
                                      overWriteOK, append,
                                      firstEvent, streamId,
                                      splitNumber,
                                      splitIncrement, streamCount,
                                      compressionType, compressionThreads,
                                      ringSize, bufferSize);
```

The baseFilename string will have the value of the environmental variable BASE_NAME, substituted for the $(BASE_NAME) part of the string along with the runType substituted for the %s, the runNumber substituted for the %x (hex format), and the split number substituted for the %03d. If BASE_NAME has the value “File”, then the first 3 split files will have the names:

```
myFile_myExperiment_1_001.ext
myFile_myExperiment_1_002.ext
myFile_myExperiment_1_003.ext
```

6.5 *Regular API for Reading Events*

Start reading an evio format file or buffer with an ***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 record numbers are sequential. (Find out about record numbers by reading through the section which describes the evio file format). There is also an optional parameter for choosing between sequential and random-access methods for reading a file.

There are two basic types of read calls. The first is random-access, and the second is sequentially in nature. 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, be aware that calling a random-access method will affect the sequential methods. The sequence will jump to the event called by event number. For example, if an application calls parseNextEvent() twice, then suddenly calls parseEvent(20), the next time parseNextEvent() is called, it will returned the 21st event, not the 3rd.

Now for a word on performance. 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("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();}
}
```

6.6 *Searching*

Most users are interested in searching for a specific event or an evio structure. To this end, evio has several built-in searches for ease of use. See the doxygen documentation for the static methods in 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 vector. In addition, listeners can be added to execute methods upon the finding of a matching structure using ***IEvioListener*** interface.

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 event for banks (not segs, tagsegs) with particular tag & num values
// using StructureFinder.
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");
    }
}

// ------------------------------------------------------------------
// Define a filter to select Segment structures with odd numbered tags.
System.out.println("Search for segments with odd numbered tags, got:");
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");
    }
}

// ------------------------------------------------------------------

// Define a custom listener
class myListener implements IEvioListener {
     public void gotStructure(BaseStructure topStructure,
                              IEvioStructure structure) {
          System.out.println("listener got structure = " + structure.toString());
     }

     // We're not parsing, so these are not used ...
     public void startEventParse(BaseStructure structure) {}
     public void endEventParse(BaseStructure structure) {}
};

// Custom search for segments with odd tags which also executes listener
System.out.println("Search all segs with odd tags, & execute listener, got:");

// Do the searching
myListener listener = new myListener();
event.visitAllStructures(listener, filter);
```

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

6.7 *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.

Evio 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:

1. after a structure is read in and filter applied while an event is being searched/parsed
2. just before an event (bank, segment, or tagsegment) is to be parsed, and
3. just after an event has been parsed.

Here again, the previous section has a simple example of doing this. Following is an example of code that uses both a listener and a filter, but this time in the context of parsing.

```
// Read some data
EvioReader reader = new EvioReader(buffer);
EventParser parser = reader.getParser();

// Define a listener for the parser
class myListener implements IEvioListener {

   // Call after structure is read in & filter applied while
   // event is being parsed
   public void gotStructure(BaseStructure topStructure,
                            IEvioStructure structure)  {
       System.out.println("GOT struct = " + structure.toString());
   }

   // Call at start of event parsing
   public void startEventParse(BaseStructure structure) {
       System.out.println("START parsing event = " + structure.toString());
   }

   // Called at end of event parsing
   public void endEventParse(BaseStructure structure) {
       System.out.println("END parsing event = " + structure.toString());
   }
};

myListener listener = new myListener();

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

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

myFilter filter =new myFilter();

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

// Now parse an event
System.out.println("Run custom filter & listener on first event:");
EvioEvent ev = reader.parseEvent(1);
```

6.8 *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 and the differences in the format of their header words. For example:

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

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

6.9 *Dictionaries*

This section describes how dictionaries can be used (refer to relevant section for more details). Define a dictionary by something like:

```
// Define xml dictionary String
String xmlDictString =
      "<xmlDict>\n" +
      "  <dictEntry name=\"myEntry\"   tag=\"1\"  num=\"1\" />\n" +
      "  <dictEntry name=\"second bank\"  tag=\"2\"  num=\"2\" />\n" +
      "  <bank name=\"parentEntry\"  tag=\"10\"  num=\"3\" >" +
      "    <leaf name=\"childEntry\"  tag=\"20\"  num=\"4\" />" +
      "   </bank>" +
      "</xmlDict>";

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

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

Once the dictionary is created, 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. 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 myName     = "myEntry";
String childName  = "childEntry";
String parentName = "parentEntry";

// Search for structures (banks, segs, tagsegs) with a particular name
List<BaseStructure> list1 = StructureFinder.getMatchingStructures(
                                    event, myName, 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's getDictionaryXML()*** and ***hasDictionaryXML()*** methods. Similarly, the ***EvioCompactReader*** has ***getDictionaryXML()***, ***getDictionary()***, and ***hasDictionary()*** methods. 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 if 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!" +
 "   </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));
```

There are also a couple of ways in which to iterate through the entries of a dictionary to see what it contains using the ***dict.getMap()*** method.

Method 1:

```
Map<String, EvioDictionaryEntry> map = dict.getMap();
Set<String> keys = map.keySet();
for (String key : keys) {
    System.out.println("key = " + key +
                       ", tag = " + dict.getTag(key) +
```

```
                        ", num = " + dict.getTag(key));
}
```

Method 2:

```
int i=0;
Map<String, EvioDictionaryEntry> map = dict.getMap();
Set<Map.Entry<String, EvioDictionaryEntry>> set = map.entrySet();
for (Map.Entry<String, EvioDictionaryEntry> entry : set) {
    String entryName =  entry.getKey();
    EvioDictionaryEntry entryData = entry.getValue();
    System.out.println("entry " + (++i) + ": name = " + entryName +
                       ", tag = " + entryData.getTag() +
                       ", num = " + entryData.getTag());
}
```

6.10 *First Event*

If the user wants the same (first) event in each split file, then simply select the ***EventWriter*** constructor that has an argument for the first event (for either file or buffer writing). An alternative method is to call one of the EventWriter's ***setFirstEvent()*** methods. If calling the method, make sure it's called before any other events are written in order to ensure that it is written to each of the split files.

6.11 *XML format events*

If the user wants to view an event in xml format, that is easily possible:

```
EvioEvent ev;
boolean asHex = true;
String xml1 = ev.toXML();
String xml2 = ev.toXML(asHex);
```

It's also possible to go in the other direction and parse a file of xml events into EvioEvent objects. In this case, if there is more than one event, the top-level xml element must be:

```
<evio-data>
```

If there is no dictionary, the events must be indicated by the xml element:

```
<event>
```

otherwise it can be any valid xml element whose value exists in the dictionary. Elements whose tag/num/type info is not in the xml may have it supplied by a dictionary entry.

```
String xml;
List<EvioEvent> list;
list = Utilities.toEvents(xml);
```

```
int maxEvents = 20, skip = 1;
EvioXmlDictionary dictionary;
boolean debug = false;
list = Utilities.toEvents(xml, maxEvents, skip, dictionary, debug);
```

Chapter 7

7 Evio Structure Agnostic API

It may sound weird to have an API which does not know about evio structures such as banks, etc. This, in fact, arose from the Java HIPO library which was incorporated into the Java evio library and eventually into the C++ evio library. This interface deals with arrays of bytes, hence there are no classes for building events. It's up to the user to deal with lower level data format.

7.1 *Writing Events*

7.1.1 Writing to file or buffer

Start writing a file or buffer with a ***Writer*** object. The user can set the maximum record size, the maximum number of events per record, the type of data compression desired, the output byte order as well as specify a dictionary among other things. Refer to the doxygen documentation for all of the possibilities.

Note that for writing to file, if the file name is not available as a constructor argument, then the method ***open()*** must be called, otherwise it's called in the constructor. When writing to a buffer, the variant of ***open()*** specifying a buffer may be called when one wants to switch buffers. If that method previously called, then ***reset()*** must be called before it can be called again.

Below is simple Java example code with comments showing how writing a file is done (C++ is very similar):

```
// Create writer
Writer writer = new Writer();
// Choose LZ4 compression
writer.setCompressionType(CompressionType.RECORD_COMPRESSION_LZ4);
// Choose file
writer.open("/home/exampleDataFile");

// Fill this array with data …
byte[] buffer;

// write data
```

```
writer.addEvent(buffer);

// Add trailer to file
writer.addTrailer(true);

writer.close();
```

7.1.2 Multithread Compression for File Writing

There is a multithreaded sibling to the Writer class called ***WriterMT.*** This is a class for the purpose of writing bytes to a file using multiple threads for compressing data. The compression computation is distributed to threads operating in parallel, the number of which is determined in the constructor (1 unless otherwise given).

At the center of how ***WriterMT*** works (and hidden from the user) is an ultra-fast ring buffer containing a store of empty records. As the user calls one of the ***addEvent()*** methods, it gradually fills one of those empty records with data. When the record is full, it's put back into the ring to wait for one of the compression threads to grab it and compress it. After compression, it's again placed back into the ring and waits for a final thread to write it to file. After being written, the record is freed up for reuse.

Below is a little more complicated Java example code with comments (C++ is very similar):

```
String filename = "myFile";
// Use little endian byte order
ByteOrder order = ByteOrder.LiTTLE_ENDIAN;
// Max number of events a record can hold.
// Use default of 1M.
int maxEventCount = 0;
// Max number of uncompressed data bytes a record can hold.
// Use default size of 8MB.
int maxBufferSize = 0;
// Use GZIP compression
CompressionType compType = CompressionType.RECORD_COMPRESSION_GZIP;
// Number of threads available to do compression
int compThreads = 2;
// Number of records held in internal ring buffer
int ringSize = 8;


WriterMT writer = new WriterMT(order, maxEventCount, maxBufferSize,
                               compType, compThreads, ringSize);
writer.open(filename);

// Place data in this array
byte[] buffer;

// Add data
writer.addEvent(buffer);

// Evio data can also be written
EvioBank bank = ….
```

```
writer.addEvent(bank);

// Add trailer which will contain the
// locations of all records in the file
writer.addTrailerWithIndex(true);

writer.close();
```

Although, technically speaking, using this API the user could deal with records directly, the record structure of the HIPO and evio format should be completely transparent to the user.

7.2 *Reading Events*

To read a file or buffer in an evio structure independent manner use the ***Reader*** class. This has many methods that can be seen in the javadoc or doxygen documentation. Dictionaries, first events, the “user header” of the file’s header, and many other quantities can be read as well as the events themselves. Following is a short Java example, notice that both sequential and random-access style API calls are available.

```
String filename = "/tmp/testData";

// Create file reader
Reader reader = new Reader(filename);

// Get the number of contained events
int nevents = reader.getEventCount();

// Print out the file header
System.out.println("file header: \n" + reader.getFileHeader().toString());

// Read any dictionary
if (reader.hasDictionary()) {
    String dict = reader.getDictionary();
}

// Sequential API to read events
while (reader.hasNext()) {
    byte[] pEvent = reader.getNextEvent();
}

// Random access API to read events
for (int i = 0; i < nevents; i++) {
   System.out.print("Get EVENT #" + (i+1));
   byte[] pEvent = reader.getEvent(i);
   int eventLen = pEvent.length;
   System.out.println(", size = " + eventLen);
}

reader.close();
```

Chapter 8

8 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).

8.1 *evio2xml*

evio2xml, part of the C library, 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] [-dtag dtag]
           [-ev evtag] [-noev evtag] [-frag frag] [-nofrag frag]
           [-max_depth max_depth]
           [-n8 n8] [-n16 n16] [-n32 n32] [-n64 n64]
           [-verbose] [-brief] [-no_data] [-xtod] [-m main_tag]
           [-e event_tag]
           [-indent indent_size] [-no_typename] [-maxbuf maxbuf]
           [-debug]
           [-out outfilenema] [-gz] filenamee
```

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. 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.

8.2 *eviocopy*

eviocopy, part of the C library, 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] [-nonum evnum] [-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.

8.3 *Xml2evio*

Xml2evio, part of the Java library, takes a file containing an xml representation of evio events and converts it into an evio format file and/or displays it on screen.

```
$ Usage: java org.jlab.coda.jevio.apps.Xml2evio -x <xml file> -f <evio file>
                     [-v] [-hex] [-d <dictionary file>]
                     [-max <count>] [-skip <count>]
          -h    help
          -v    verbose output
          -x    xml input file name
          -f    evio output file name
          -d    xml dictionary file name
          -hex  display ints in hex with verbose option
          -max  maximum number of events to convert to evio
          -skip number of initial events to skip in xml file

          This program takes evio events in an xml file and
          converts it to a binary evio file
```

Chapter 9

9 Java Evio Event-Viewing Gui

In the JEventViewer-2.0.jar file, there is a graphical user interface for looking at EVIO format files event-by-event. This viewer has the capability of grabbing ET events and cMsg messages and parsing them as evio data. It can also look at any file as a list of 32 bit integers – very useful for debugging. To run it either run:

```
java org/jlab/coda/eventViewer/EventTreeFrame
```

or run the following script contained in the package.

```
jeviodump
```

For this to work the JEventViewer-2.0.jar and the latest jevio jar files must be in your CLASSPATH. If you want to look at ET and cMsg data, then the ET and cMsg jar files must be in the CLASSPATH as well. For further documentation look in the JEventViewer package.

Chapter 10

10 Evio File and Record Format

Following are descriptions of both the old format and the new.

10.1 *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:

BLOCK HEADER

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

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

1. 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).
2. The block number is an id # used by the event writer.
3. 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.
4. The start is the offset in words to the first event header in block relative to the start of the block.

5. 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.
6. The version is the current evio format version number (1,2, or 3).
7. The reserved 1 is unused.
8. 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.

10.2 *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:

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

1. The block length is number of 32 bit words in the block (including itself). In general, this will vary from block to block.
2. The block number is an id # used by the event writer.
3. 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.

4. 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.
5. Reserved 1 is used in the CODA online to store the CODA id of the data source if bits 10-13 of the bit-info word show a ROC Raw type.
6. 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.
7. Reserved 2 is unused.
8. The magic number is the value 0xc0da0100 and is used to check endianness.

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, Other = 15
14	= 1 if the next event (non-dictionary) is a “first event” to be placed at the beginning of each written file and its splits
31-15	unused

Bits 8-14 are only useful for the CODA online use of evio. That’s because only a single CODA event ***type*** is placed into a single (ET, cMsg) buffer, and each user or control event has its own buffer as well. That buffer is then parsed by an EvioReader or EvioCompactReader object. Thus, all events will be of a single CODA type.

10.3 *Version 6*

A big factor for introducing yet another evio version was the desire to compress data in each block (now called a record). The HIPO format, in use in Jefferson Lab's Hall B, already had the capability to compress data. In order to avoid proliferation of data formats, HIPO was merged with evio along with much of the code to handle data compression. This has added a great deal of complexity to the file and record headers as one can see following.

FILE FORMAT

Below is a diagram of how an HIPO/evio6 file is laid out. Each file has its own file header which appears once, at the very beginning of the file. It is always uncompressed. That is followed by an optional, uncompressed index array which contains pairs of 4-byte integers. Each pair, one pair per record, is the length of a record in bytes followed by the event count in that record. Generally, the index array is not used with evio files.

That, in turn, is followed by an optional, uncompressed user header which is just an array of data to be defined by the user. This array is padded to a 4-byte boundary. To be more specific, it holds user data in HIPO files, but in evio files the user header will only store any dictionary or first event provided to the file writer. The dictionary and first event are placed into a record and written out as the user header. More on records later. Finally, any index array and user header is followed by any number of records including the last record which may be a trailer. More on the trailer later.

FILE HEADER

First, let's look at the file header (unless specified, each row signifies 32 bits):

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

File Type ID
File Number
Header Length
Record Count
Index Array Length
Bit info | Version
User Header Length
Magic Number
User Register, 64 bits
Trailer Position, 64 bits
User Integer 1
User Integer 2

1. For a HIPO file, the file type ID is 0x43455248, for an Evio file, it's 0x4556494F. This is asci for HIPO and EVIO respectively.
2. The split number is used when splitting files during writing.
3. The header length is the number of 32 bit words in this header - set to 14 by default. Even though, theoretically, it can be changed, there are no means to do so through the evio library.
4. The record count is the number of records in this file.
5. The optional index array, which follows this file header, contains all record lengths in order. Each length is an unsigned 4-byte integer in units of bytes. This file header entry is the length of the index array in bytes. For evio files this will be 0 as the index array is not used in any evio library.
6. The version is the current evio format version number (6) and takes up the lowest 8 bits. The bit info word is used to store the various useful data listed in the table below.
7. The user "header" will hold any dictionary and first event provided to the file writer. This file header entry is the length of the user header in bytes.

8. The magic number is the value 0xc0da0100 and is used to check endianness.
9. The user register is a 64-bit register available to the user. This is not used by evio and always set to 0.
10. The trailer position is a 64-bit integer containing the byte offset from the start of the file to the trailer (which is a type of record). This may be 0 if the position was not available when originally writing or if there is no trailer.
11. The user integer #1 is just that, a 32-bit integer available to the user. This is not used by evio and always set to 0.
12. The user integer #2 is just that, a 32-bit integer available to the user. This is not used by evio and always set to 0.

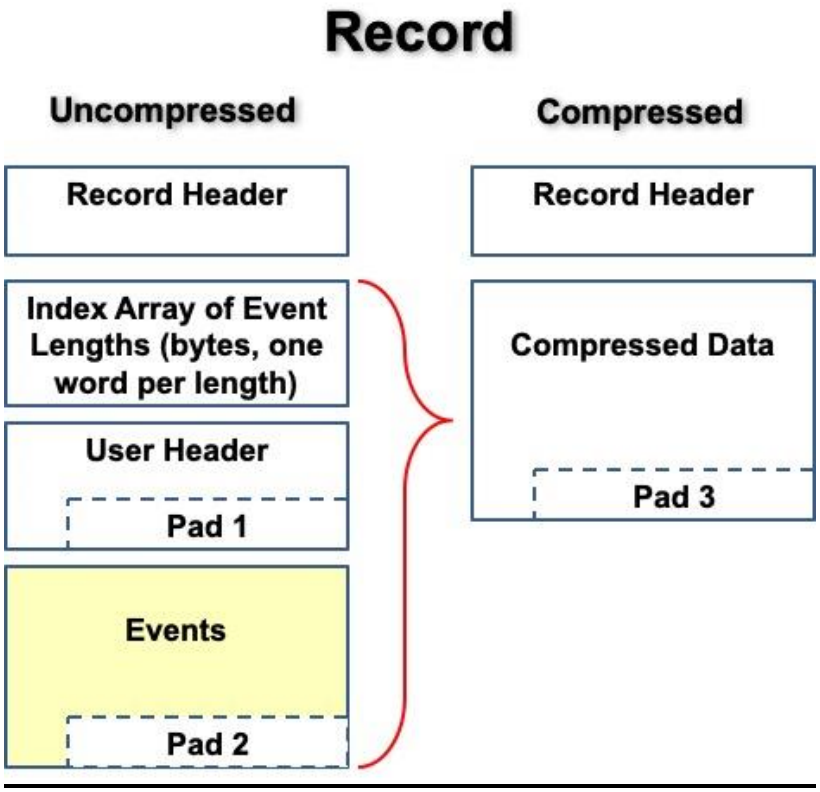
FILE HEADER'S BIT INFO WORD

Bit # (0 = LSB)	Function
0-7	Evio Version # = 6
8	= 1 if dictionary is included
9	= 1 if file has a "first" event – an event which gets included with every split file
10	= 1 if file has trailer containing an index array of record lengths
11-19	reserved
20-21	pad 1
22-23	pad 2
24-25	pad 3, always 0
26-27	reserved
28-31	What type of header is this? 1 = Evio file header, 2 = Evio extended file header, 5 = HIPO file header, 6 = HIPO extended file header

A note on the "evio extended file" header mentioned in the bitinfo table immediately above. The idea is to extend the header by adding extra words to its end. This has never been implemented. If the header was to be extended, the extra words would be user-defined integers. The third word, the header length, would need to be changed from it's normal value of 14 to include the extra words.

RECORD

After the file header, index array, and user header, the file is divided into **records**. Following is the layout of a record:

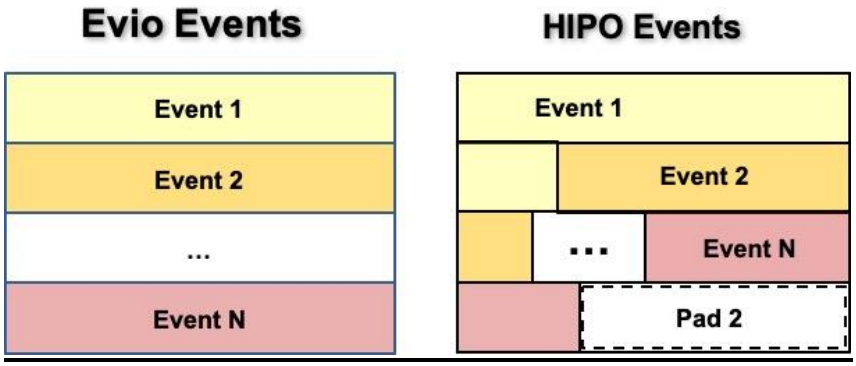


Each record has a header. The header is followed by an index of events lengths – one word per length in the unit of bytes. This index is **not** optional.

It is followed by an optional user header which functions in the same manner as the one in the file layout. It is an array of data defined by the user. This array is padded to a 4-byte boundary. For the evio format, if writing to a file, any dictionary and/or first event are exclusively placed in the file's user header and nothing is ever placed in a record's user header. If writing to a buffer, any dictionary and/or first event are exclusively placed in the user header in the format of a record (in other words, a record within a record). The EventWriter class, when writing to a buffer only writes one record and so that will contain the dictionary/first event. If using the Writer class, it's data format agnostic and any user data may be written with any record as part of the user header.

Any index array and user header is followed by any number of events. These events look differently depending on whether you have evio or HIPO format. An evio event always ends on a 4-byte boundary which means that pad2 will always be 0. In the HIPO world, events can be of any size. See the graphic below.

If data compression is being used, the record header will not be compressed allowing a quick scan of the file without any decompression needing to take place. When reading, data is decompressed record by record as needed to access particular events.



RECORD HEADER

The fields of the record header are seen below (unless specified, each row signifies 32 bits):

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

Record Length
Record Number
Header Length
Event Count
Index Array Length
Bit info | Version
User Header Length
Magic Number
Uncompressed Data Length
CT | Compressed Data Length
User Register 1, 64 bits
User Register 2, 64 bits

1. The record length is number of 32 bit words in the record (including itself). In general, this will vary from record to record.

2. The record number is an id # used by the event writer. Can be used by reader to ensure data is received in the proper order.
3. The header length is the number of 32 bit words in this header - set to 14 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 evio libraries.
4. The event count is the number of events in this record - always integral.
5. The index array length is the length, in bytes, of the following index of event lengths.
6. The version is the current evio format version number (6) and takes up the lowest 8 bits. The bit info word is used to store the various useful data listed in the table below.
7. The user “header” is just a user-defined array which may contain anything. This record header entry is the length of the user header in bytes. If writing to a buffer and if there is a dictionary / first event, they will be stored in the user header and the length of that data will be stored here. If writing to a file, there is no user header data in a record so this entry will be 0.
8. The magic number is the value 0xc0da0100 and is used to check endianness.
9. The uncompressed data length is the padded length, in bytes, of the entire uncompressed record.
10. The highest 4 bits signify the type of compression used in the record (see table below). The other bits contain the padded length of the compressed data in 32-bit words.
11. User register #1, 64 bits.
12. User register #2, 64 bits.

TYPE OF COMPRESSION

Value	Compression Type
0	No compression
1	LZ4, fast
2	LZ4, best compression
3	gzip

RECORD HEADER'S BIT INFO WORD

Bit # (0 = LSB)	Function
0-7	Evio Version # = 6
8	= 1 if dictionary is included (first record only)
9	= 1 if a first event is included (first record only)
10	= 1 if this record is the last record in file, buffer, or network transmission
11-14	type of events in record: ROC Raw = 0, Physics = 1, Partial Physics = 2, Disentangled Physics = 3, User = 4, Control = 5, Other = 15
11-19	reserved
20-21	pad 1
22-23	pad 2
24-25	pad 3
26-27	reserved
28-31	What type of header is this? 0 = Evio record header, 3 = Evio file trailer, 4 = HIPO record header, 7 = HIPO file trailer

Bits 8-14 are only useful for the CODA online use of evio. That's because, for CODA online, only a single CODA event ***type*** is placed into a single record, and each user or control event has its own record as well. Thus, all events will be of a single CODA type.

Following is a graphic displaying how the bit info word is laid out for both the file and record headers.

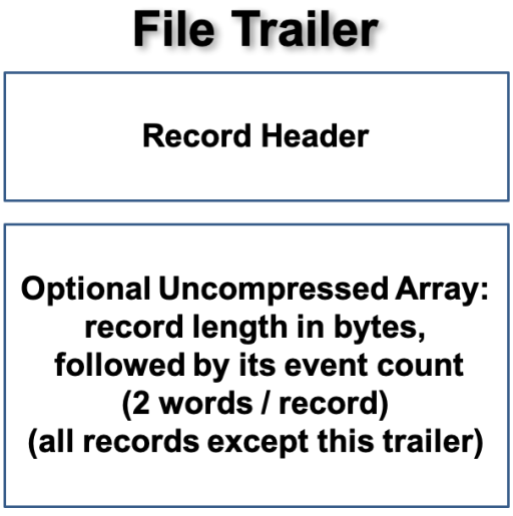
File/Record Headers, Bit Info / Version Word

Value	Header Type
0	Evio record
1	Evio file
2	Evio extended file
3	Evio file trailer
4	HIPO record
5	HIPO file
6	HIPO extended file
7	HIPO file trailer

Bits	Padding
25-24	Pad 3
23-22	Pad 2
21-20	Pad 1

TRAILER

In an evio file, the ending record can, but does not have to be, a regular record. It can also take the form of a trailer. The trailer is a regular record header optionally followed by an uncompressed array of pairs of a record length (unit of bytes, 32-bit integer) and an event count – one pair for each record. This array is put in the place of the non-optional index array of events lengths that is part of the normal record header. See below.



TRAILER'S HEADER

In either case, bit #10 of the bit info word in the record header needs to be set – indicating that this is the last record. In the case of the trailer, bits 28-31 (seen in the table above) will indicate that this is a trailer. For an evio trailer, the value in those bits is 3. Also, the version number is placed in that same word. Everything after the magic number is 0.

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

Record Length
Record Number
14
0
Index Array Length
0x 30 00 02 06
0
0xcoda0100
0
0
0
0

FILE READING

When reading a file, the reader will first read the file header. If available, it can get a list of record lengths from that header and from there calculate record positions.

If the file header's index array is non-existent, the trailer position can be read. If it's a valid value, the reader can jump to it, read the trailer, read the trailer's index, and from there calculate the position of each record.

If no record length information is available in either the file header or the trailer, the reader will scan the whole file from beginning to end to obtain it.

Chapter 11

11 EVIO Data Format

11.1 *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

MSB 32 bits LSB

length			
tag	**pad**	**type**	**num**
Bits: 16	2	6	8

SEGMENT HEADER

tag	**pad**	**type**	**length**
Bits: 8	2	6	16

TAGSEGMENT HEADER

tag	**type**	**length**
Bits: 12	4	16

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 int
0xc	TAGSEGMENT
0xd	SEGMENT
0xe	BANK
0xf	Composite
0x10	BANK
0x20	SEGMENT
0x21	Hollerit*
0x22	N value*
0x23	n value*
0x24	m value*

*this type is only used internally for composite data

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.

11.2 *Changes from Versions 1-3*

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.

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.

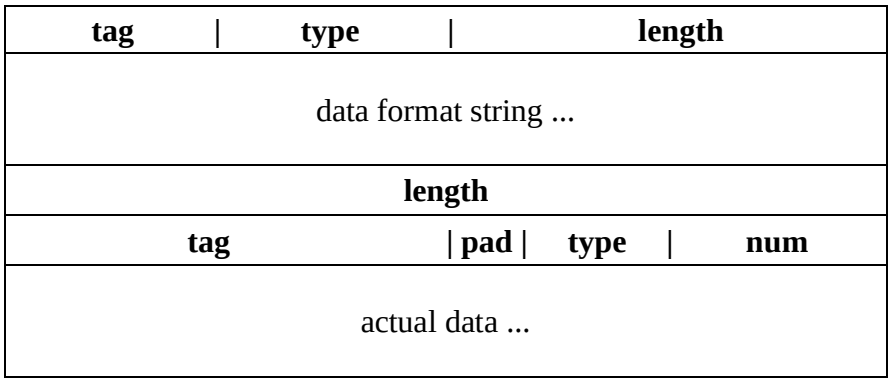
The pad in the BANK and SEGMENT types indicates the number of bytes used for padding to 32 bit boundaries when type indicates 8 or 16 bit integers (type = 0x4, 0x5, 0x6, or 0x7). For 16 bit types pad will be 0 or 2 while for the 8 bit types it will be 0-3. Unlike previous versions, this allows EVIO to 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.

11.3 *Composite Data Type*

11.3.1 General Type Info

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



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 as 32-bit int,
n	Multiplier as 16-bit int
m	Multiplier as 8-bit int

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

i,L,2(s,2D,mF)

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, a multiplier of 2 (32 bit int) of everything inside the parentheses which ends up being an unsigned short, 2 doubles, an 8-bit multiplier, multiplier number of 32-bit floats, an unsigned short,

2 doubles, an 8-bit multiplier, 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, 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.

11.3.2 Creating Events with Composite Data

Handling composite data is best done through the ***CompositeData*** and its contained ***CompositeData.Data*** classes. Although the following example uses Java, it has an identical counterpart in C++ as well. Generally one defines a CompositeData.Data object and uses that to create an instance of the CompositeData class which can , in turn, be used when building an event. The building can be done with either the EventBuilder or CompactEventBuilder class. It's easiest to look at a couple examples to see how it's done.

```
// Format to write a N shorts, 1 float, 1 double a total of N times
String format1 = "N(NS,F,D)";

// Now create some data
CompositeData.Data myData1 = new CompositeData.Data();
// First N (describing # of groups) is 2
myData1.addN(2);
// Next N (describing # of shorts) is 3
myData1.addN(3);
// Since N=3, there are 3 shorts and we must provide them all
myData1.addShort(new short[]{1, 2, 3}); // use array for convenience
myData1.addFloat(1.0F);
myData1.addDouble(Math.PI);
// Since the first N says there are 2 groups, this is N (1) to describe the
// number of shorts in the second group
myData1.addN(1);
// Add the single short in the second group
myData1.addShort((short) 4); // use array for convenience
myData1.addFloat(2.0F);
myData1.addDouble(2. * Math.PI);
```

The next example is tricky because it uses strings. An array of strings in evio is described by the DATACHAR8 data type and we don't immediately know how long it is because evio does some internal packaging of the strings into a form which falls on a 4 byte boundary. Thus, we don't know the format right off the bat but need to do a little work to figure it out.

```
// We want a format for writing an unsigned int, unsigned char,
// and N number of M (to be found) ascii characters & 1 64-bit int.
// We need to wait before we can create this format string because we
// don't know yet how many String characters (M) we have.
// The format will be something like  "i,c,N(Ma,L)";

// Now create some data
CompositeData.Data myData2 = new CompositeData.Data();
// Add unsigned int
myData2.addUint(21);
// Add unsigned char
myData2.addUchar((byte) 22);
// Group of 1 (string + long)
myData2.addN(1);
// Now define our ascii data
String s[] = new String[2];
s[0] = "str1";
s[1] = "str2";
// Find out what the composite format representation of these strings is
String asciiFormat = CompositeData.stringsToFormat(s);
// Full format is:
String format2 = "i,c,N(" + asciiFormat + ",L)";
// Add string data now
myData2.addString(s);
// Finally the long
myData2.addLong(24L);
```

We now have two composite data items and we can add them to the event we're creating.

```
// Create CompositeData array
CompositeData[] cData = new CompositeData[2];
try {
  // fTag is the tag in tagsegment header describing format string.
  // dataTag and dataNum are tag and num of bank describing actual data
   cData[0] = new CompositeData(format1, fTag1, myData1, dataTag1, dataNum1);
   cData[1] = new CompositeData(format2, fTag2, myData2, dataTag2, dataNum2);
}
catch (EvioException e) {
  e.printStackTrace();
}

// Using a CompactEventBuilder do something like:
builder.openBank(tag, num, DataType.COMPOSITE);
builder.addCompositeData(cdata);
builder.closeStructure();

// Using an EventBuilder do something like:
```

```
EvioBank bankComps = new EvioBank(tag, DataType.COMPOSITE, num);
bankComps.appendCompositeData(cdata);
builder.addChild(bankBanks, bankComps);
```

Chapter 12

12 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 evio characteristics, for example a tag and a num. The xml protocol was used to accomplish this. The following gives the different xml formats used by the different versions of evio.

12.1 *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.

12.1.1 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.

12.1.2 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.

12.2 *Evio versions 4 and later*

The C/C++ and Java dictionaries now have identical formats.

12.2.1 Changes

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.

12.2.2 Element and Attribute Names

In the old 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, unint32, long64, ulong64, short16, ushort16, char8, uchar8,
charstar8, float32, double64, bank, segment, tagsegment, composite,
unknown32
```

12.2.3 Tag & Num Values

Specifying tag and num values can be done in multiple ways. Look at the dictionary following:

```
<xmlDict>
   <dictEntry name="TagNum"          tag="1"     num="1" />
   <dictEntry name="TagRange"        tag="5-9" />

   <dictEntry name="Tag_%t_A"        tag="10" />
   <dictEntry name="Tag_%t_B"        tag="20"    num="2" />

   <dictEntry name="TagNum_%n"       tag="30"    num="3" />
   <dictEntry name="TagNum_%n"       tag="40"    num="4-6" />

   <dictEntry name="Tag_%t_Num_%n"   tag="50"    num="9" />
   <dictEntry name="Tag_%t_Num_%n"   tag="50"    num="10-11" />
</xmlDict>
```

There are some constructs that are new:

5. The first entry associates a single tag and a single num value with that entry. This entry matches only structures with that exact tag & num.

6. The second entry defines a range of tags. This matches any structure with a tag in the listed range, inclusive. Note that entries of this type cannot simultaneously define a num or range of nums, i.e. nums are disregarded.

7. The third entry has a ***%t*** in the name which gets substituted with the value of the tag. Thus, it gets entered into the dictionary as ***Tag_10_A.*** It associates the entry with a single tag and matches any structure with that tag. Note that

entries of this type cannot define a range of tags. Thus, <dictEntry name="tagRange%t" tag="5-9" /> is not legal.

8. The fourth entry is similar to the third, except that it also associates a single num with the tag and gets entered into the dictionary as ***Tag_10_B*** .

9. The fifth entry substitutes the num value for ***%n*** in the name. Thus, the new entry, named ***TagNum_3*** is associated with tag=30 and num=3.

10. The sixth entry gets multiplied into several entries, one for each num substituted into the name with the final names, ***TagNum4***, ***TagNum5***, and ***TagNum6***.

11. The seventh entry has substitutions in its name, ***Tag_50_Num_9***, for for both tag and num values but is otherwise equivalent in function to the fifth entry.

12. And final entry is the same the the seventh except that it gets multiplied into 2 entries since a num range is defined: ***Tag_50_Num_10,*** and ***Tag_50_Num_11.***

A word about ranges. The purpose of a range of nums is merely to save space and effort. Many actual dictionary entries, one per num value, are generated with a single ranged entry. On the other hand, the purpose of a range of tags is different. It serves to group multiple evio structures together under a common label. This may be somewhat confusing so here are the rules:

- If a tag range is defined, no num value or num range is allowed
- If a num range is defined, it must be used in conjunction with ***%n*** in the entry's name
- A ***%t*** substitution cannot be made if a tag range is defined
- Duplicate entries are ignored.

12.2.4 Types of Dictionary Entries

As we've already seen, 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).

On the other hand, the elements ***bank*** and ***leaf*** are used for describing hierarchical evio 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"/>
```

Notice that a period, “.”, separates parent from child in the hierarchy. 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='F,D,Ni' >
                 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.

12.2.5 Matching Priorities

Given a dictionary, how are matches made and prioritized? Matches are made with the in the dictionary in the following order:

1. The first entry specifying both a single tag and a single num matches a bank or event with the same tag and num.
2. The first entry specifying only a single tag matches any structure with the same tag.
3. The first entry specifying a range of tags matches any structure with a tag in that range.

12.2.6 Pretty Printing

Dictionary entries without the ***num*** attribute may be defined in order to beautify any printed output:

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

For example, say the 2 dictionary entries above are the only ones defined for tag = 1. Now, if an evio data file is being printed and contains a bank with tag = 1 and num = 2, what this does is assigned the name “GeneralTag” to such a bank whose specific tag/num pair has no corresponding dictionary entry.

12.2.7 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 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 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 Java toXml() method of a dictionary is called, only the dictionary portion of the original, full xml document is returned as a String.

12.2.8 Dictionary Class

So far, this chapter has described the xml format of the dictionary. However, in both C++ and Java, the dictionary is also an object of the ***EvioXMLDictionary*** class (which takes the xml string as a constructor arg). This class has methods to directly access the information from having parsed the xml. See doxygen and Javadoc docs for details as well as the follow C++ section and Java section.

The dictionary in object form can also be used in searches by means of the ***StructureFinder*** and other classes. Again, see doxygen and Javadoc docs for details.

Appendix

A

A. Third Party Software Packages

A.1 Spack

Spack is a software package manager for linux, macOS, and supercomputers that makes installing scientific software much easier. To allow evio to be used by spack, it has a corresponding **package.py** config file. This is available in the https://github.com/JeffersonLab/epsci-spack repository on github.

A.2 Disruptor

If one is not using https://github.com/JeffersonLab/disruptor (forked from the original) but is instead using the original distribution at https://github.com/LMAX-Exchange/disruptor, be aware that the jar file made available as part of the evio distribution has had one class added and 5 or 6 of them slightly modified. Thus, in the src/main/java/com/lmax/disruptor directory, one must add the ***SpinCountBackoffWaitStrategy.java*** file to any distribution taken directly from the original distribution, and one must make slight additions to YieldingWaitStrategy.java, SingleProducerSequencer.java, Sequenced.java, RingBuffer.java and MultiProducerSequencer.java (each change marked by a string containing “Timmer” or “timmer”). If existing, the src/test/java/com/lmax/disruuptor/RingBufferWithAssertingStubTest.java file was also modified.

Similarly, in C++, if one is not using https://github.com/JeffersonLab/Disruptor-cpp (forked from the original) but is instead using the original distribution at https://github.com/Abc-Arbitrage/Disruptor-cpp, in the Disruptor directory, the files **SpinCountBackoffWaitStrategy.h** and **SpinCountBackoffWaitStrategy.cpp** must be added. Also, extensive changes have been made to **CMakeLists.txt** and **Disruptor/CMakeLists.txt**, so these need to be replaced as well.

Appendix

B

B. Alternate ways to generate documentation

B.3 Read the docs

In order to host the user's guide on the readthedocs website, it must first be transformed into the reStructuredText (rst) format. This is done by using the ***pandoc*** program available from https://pandoc.org. Follow instructions from that website in order to install it. On the Mac it's as simple as:

```
brew install pandoc
```

The ***brew*** command is available from https://brew.sh and is a package manager for the Mac. Once installed, the following command will do the transformation from word to rst:

```
pandoc evio_UsersGuide.docx -f docx -t rst -s -o evio_UsersGuide.rst
```

Sphinx is a Python software package which is designed to output documentation in various formats from input in rst format. This is why the user's guide was transformed into rst format – so that sphinx could work its magic. Sphinx can be installed with the command:

```
pip install sphinx
```

If doxygen generated output is to be included along with the user's guide, a few more things need to be done. The ***breathe*** and ***exhale*** plugins for sphinx are necessary to incorporate doxygen comments into the final product. Breathe takes the xml format doxygen output file and forms it into rst format. This means that the doc/DoxyfileCC and doc/DoxyfileC files need to be modified to produce xml output. Exhale takes the breathe output and makes rst which allows one to navigate between classes and files. These packages can be installed by calling:

```
pip install breathe
pip install exhale
```

However, as in the case with doxygen, dealing directly with sphinx and breathe is unnecessary. The readthedocs site is already linked to evio's github location and can automatically generate docs using its own sphinx and breathe. At least this is

true theoretically. In practice, doxygen must be configured to produce an xml format output file. Unfortunately, it produces an xml file in which the last element does not match its first element. This must be corrected by hand in order for the xml parsing to succeed.

Producing the final documentation by hand requires the following steps:

1. run doxygen
2. edit the produced index.xml file so it can be parsed
3. setup the sphinx configuration file, source/conf.py
4. create the content to be finally displayed, in index.rst
5. create the html file through sphinx by calling:

```
make html
```

6. look at the produced html file, build/hml/index.html, with a browser

B.2 GitHub Pages

It's also possible to host documentation through a github repository directly. To do this:
Check doxygen generated html files into the repository.
Create an index.md file with markup language directions on what to display
In the repository's setting, set the GitHub Pages area to point to this file.