EVIO – A Lightweight Object-Oriented I/O Package

Elliott Wolin, D Abbott, V Gyurjyan, G Heyes, E Jastrzembski, D Lawrence, C Timmer

Thomas Jefferson National Accelerator Facility, Newport News, VA 23606

Introduction

EVIO (EVent I/O) is a lightweight C++ package (Java version under development) that provides a simple tree-structured container for experimental data, and the ability to easily marshall data to and from disk. Endian conversions are handled automatically. EVIO additionally provides simple but powerful mechanisms to create, query, and manipulate in-memory data trees.

When to Use EVIO

EVIO is suitable whenever your data can be represented by a tree structure of leaf and container nodes, where leaf nodes contains arrays of primitive types, and container nodes contain other nodes, but not data.

EVIO and XML

The EVIO data model maps directly to XML, and EVIO utilities can easily transform between binary on-disk, in-memory, and ASCII XML formats. It borrows many ideas from the XML Document Object Model (DOM).

EVIO and the STL

The EVIO package uses templates and the Standard Template Library (STL), and its data query model is very STL-like in that it is based on iterators, algorithms, function objects, etc.

EVIO Development History

EVIO was originally developed for use in high-speed DAQ systems at JLab, and the buffer manager component has been in use for over a decade. The object-oriented features described here are a recent development that extends the utility of EVIO to cover the entire data analysis chain.

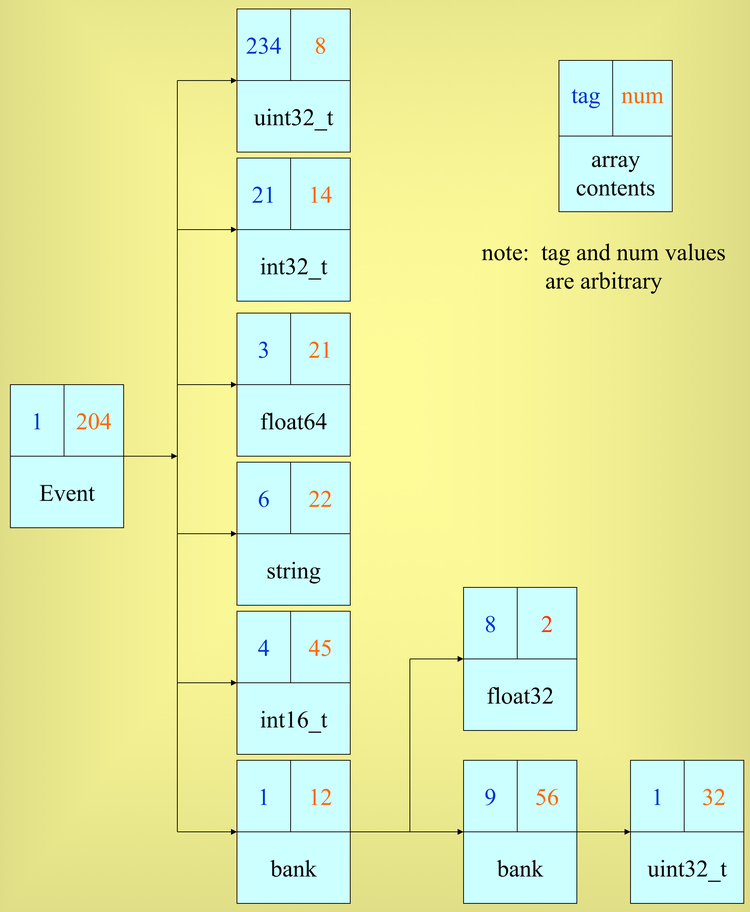
The EVIO Data Tree

EVIO data trees consist of leaf nodes (or banks), containing data, and container nodes (or banks), containing other nodes (which can then hierarchically contain other nodes or data).

Nodes (banks) are labeled with a 16-bit integer tag and an 8-bit num. Two-word headers are used on disk, and a compact 1-word header format is supported that stores less bits for tag and does not store num.

Leaf nodes contain arrays of the primitive types: int8_t, uint8_t, int16_t, …, int64_t, uint64_t, float32, and float64, or a string.

Example Data Tree



Query and Modify Tree

```
// get lists of pointers to various nodes in event tree using built-in and
//    user-defined selection function objects

evioDOMNodeListP fullList = event.getNodeList();                    // all nodes
evioDOMNodeListP fList    = event.getNodeList(typeIs<float>());     // holds floats
evioDOMNodeListP t5List   = event.getNodeList(tagEquals(15));       // has tag=15
evioDOMNodeListP myList = event.getNodeList(myFilter);              // user defined

 // apply user-defined processing function to all nodes in full list
for_each(fullList->begin(), fullList->end(), myProcessingFunction);
```

```
// simple filter function

bool myFilter(const evioDOMNodeP pNode) {

   return((pNode->tag==2)&&(pNode->num==9));
}
```

```
// simple processing function

void myProcessingFunction(const evioDOMNodeP pNode) {

  cout << "   tag is  " << pNode->tag  << ",   num is   " << pNode->num << endl;
  cout << pNode->toString() << endl;      // print node in XML
}
```

```
// get data vectors from all nodes containing floats and print

evioDOMNodeList::const_iterator iter;
for(iter=fList->begin();  iter!=fList->end(); iter++) {

    const vector<float> *fvec = (*iter)->getVector<float>();
    for(int i=0; i<fvec->size(); i++) cout  << (*fvec)[i] << endl;
}
```

```
// get list of all children of root node (same for any other container node)

evioDOMNodeList *clist = event.root->getChildList();
cout << "Root node child list length is " << clist->size() << endl;
```

```
// tree building, node and data manipulation

// create a container node using static factory method and add to tree
evioDOMNodeP cnode = evioDOMNode::createEvioDOMNode(tag=3, num=7);
event << cnode;

// create a leaf node, fill with data from vector<int> ivec1
evioDOMNodeP lnode =
                    evioDOMNode::createEvioDOMNode(tag=2, num=6, ivec1);

// hang lnode and more new leaf nodes off cnode (note dereferencing)
*cnode << lnode
       << evioDOMNode::createEvioDOMNode(tag=8, num=1, dbuf, 10)
       << evioDOMNode::createEvioDOMNode(tag=8, num=2, fvec);

// replace lnode data with data from vector<int> ivec2
lnode->replace(ivec2);

// add some more data from vector<int> ivec3
*lnode << ivec3;
```

Tree Creation and I/O

```
// create event tree and root node

int16_t  tag;
int8_t  num;
evioDOMTree event(tag=1, num=2);

// add banks to event tree in a single level below the root node
event.addBank(tag=2, num=9,  ivec);                    // vector<int> ivec
event.addBank(tag=3, num=10, ibuf,  len=8);            // int ibuf[8]
event << evioDOMNode::createEvioDOMNode(tag=7, num=14, ivec);
```

```
// write event tree(s) to disk

  // open binary file for writing
  evioFileChannel file("fakeEvents.dat","w");
  file.open();

  //  write out event tree
  file.write(event);

  //  write out other event trees…

  // close file
  file.close();
```

```
//  read and process events

  // open binary file for reading
  evioFileChannel file("fakeEvents.dat","r");
  file.open();

  // loop over all events in file
  while(file.read()) {

    // create event tree from binary event in fileChannel object
    evioDOMTree event(file);

    cout << event.toString() << endl;       // print event in XML
  }

  // close file
  file.close();
```

Example Tree in XML

```
<event   format="evio" count="1" content="bank" data_type="0x10“
              tag="1" num="204">

    <uint32_t data_type="0x1" tag="234" num="8">
        0xffffffff    0xfffffffe    0xfffffffd    0xfffffffc    0xfffffffb
        0xfffffffa    0xfffffff9    0xfffffff8    0xfffffff7    0xfffffff6
    </uint32_t>

    <int32_t data_type="0xb" tag=“21" num=“14">
              -1          -2          -3          -4          -5
              -6          -7          -8
    </int32_t>

    <float64 data_type="0x8" tag="3" num=“21">
       -1.0000000000000000000e+01  -2.0000000000000000000e+01
       -3.0000000000000000000e+01  -4.0000000000000000000e+01
    </float64>

    <string data_type="0x3" tag="6" num=“22">
<![CDATA[
hello world
]]>
    </string>

    <int16_t data_type="0x4" tag="4" num=“45">
          1       2       3       4       5       6       7       8
          9      10      11      12      13      14      15      16
    </int16_t>

    <bank content="bank" data_type="0x10" tag="1" num="12">
      <float32 data_type="0x2" tag="8" num="2">
           1.000000    2.000000    3.000000    4.000000    5.000000
           6.000000    7.000000    8.000000    9.000000   10.000000
      </float32>
      <bank content=”bank" data_type="0x10" tag=”9" num=”56">
        <uint32_t data_type="0x1" tag=”32">
              0x1         0x2         0x3         0x4         0x5
        </uint32_t>
      </bank>
    </bank>
<event>
```

Advantages

The EVIO package is fast and has a small footprint. The on-disk format is very compact, and compression is generally not needed.

The on-disk format is simple, and event trees can be written directly to disk in C without the need to create an in-memory event tree with subsequent serialization to disk. This is typically done in DAQ systems where speed is essential and trees are fairly simple.

For all other situations the C++ interface, as described on this poster, should be used. Thus the EVIO package can be used seamlessly at all levels of data taking and analysis.

EVIO is written in C++ and standard C (buffer I/O component), and compiles and runs on all Unix platforms we have available to us (many Linux variants and Solaris, multiple compilers). Please contact us if you would like to use EVIO on MS Windows.

Conclusions

The EVIO package performs fast and efficient I/O between a compact on-disk binary representation of experimental data and an in-memory tree-structured container. It further provides a rich set of facilities for creating and manipulating in-memory data. EVIO is suitable for all stages of experimental data storage and processing, from high-speed data acquisition, to monte-carlo simulation, to data reduction, to final DST analysis.

The C++ version is complete and can be downloaded from the FTP site. A Java version is under development.

Only a few EVIO features are shown on this poster. A full description can be found in the User's Manual on the FTP site.

Downloads

Download and give EVIO a try! You can get your free copy today at ftp://ftp.jlab.org/pub/coda/evio/2.0

Elliott Wolin, (757) 269-7365,
wolin@jlab.org