BinX Library Detailed Design Document

```
Design issues to be resolved:
    1.    Naming of file (dataFile) and typedef (typeDefinition) elements in schema
    2.    Unions
    3.    Variable sized and large structs.
    4.    Bit order (compiler??)
    5.    Multi file arrays
    6.    Void fields
    7.    Bit types (can't read bit from file)
    8.    Chars in Java are 16 bit Unicode
    9.    Fortran arrays have funny indices.
```

Martin Westhead 24-7-02
Daragh Byrne 16-08-2002

The purpose of this document is to:

- Specify the requirements of the BinX library
- Outline the design of the library
- Outline the design and use of the API it will provide
- Indicate how the library will be implemented in C++

Terms

BinX XML file	XML file that conforms to the BinX Schema used to describe a binary data file referred to as a BinX data file.
BinX Data file	Binary Data file which has a corresponding BinX description in XML.
Configuration	A particular combination of bit and byte order and blocksize. Applies to a field in a data file.

BinX Library Requirements

The BinX library must have the basic functionality to:

1. Provide straightforward program read access to data in any binary file described in BinX.
2. Provide the ability to programmatically describe the contents of a file and write out the corresponding BinX XML and data files.
3. Provide sufficient functionality that the library can serve as a base for flexible BinX applications.

In addition there are the following requirements

1. Efficiency (memory and CPU)
2. Access to raw data (bytes in file)
3. Access to data as structured fields
4. Access to additional BinX info (Field name etc.)
5. Static C interface.
6. Portability (Windows, Solaris, Linux at least)
7. Compile time configuration OR run time initialisation of native representation (what is the local bit/byte ordering)
8. Thread safety (threaded file I/O)

Use cases

General Considerations

The design of this library is undertaken with specific uses in mind. This section aims to describe some of these uses, and define how a programmer will use the BinX library.

A BinX data file consists of a sequence of **data elements**. Note that some confusion may arise between data elements and XML elements in the BinX XML document. The API provided by this library operates by building a representation of these elements in memory. The representation is created by reading a BinX XML file in the case of reading data, or building it using other API calls in the case of writing. This representation provides information about element size, element configuration and element metadata, such as name, to the API internals.

In many cases the programmer will be writing applications to deal with a specific class of files with a known structure. The programmer will know the names and types of the main structures in the file, and will have an idea of the meaning of the data. The BinX library must allow transparent, easy access to the data via the library. The library must also allow the programmer to build a representation of a data file in memory and write data corresponding to this representation. In this case the corresponding XML file is also written. These kind of sequential read and write operations are used in many applications already and this use of BinX in this manner is therefore justified. The benefit is that the data gains cross system portability accompanied by a structural description.

In other cases, the programmer will be writing general applications that deal with arbitrary BinX files. An example of this would be a general purpose BinX data browser. Another example would be an application that transforms the structure and configuration (given bit and byte order and block-size etc) of a BinX data file to that specified by a different BinX XML file (it is envisaged that this would be carried out using some form of XSLT transformation). These cases are examples of using BinX as a data manipulation and transformation tool.

There follow a number of specific use cases, each of which illustrate a different aspect of using the library.

Using BinX to read data when structure is roughly known

The following steps describe the basic read mechanism in BinX.

1. The BinX XML file is read and a representation is built in memory. This information is represented by an instance of BxDataset.
2. The programmer creates a BxDatasetReader object, initialising it with a pointer to the instance of BxDataset.
3. The programmer sets the reader to read a particular file referenced in the dataset.
4. The reader is asked for the values of data fields sequentially.

This is the approach used by the programmer when the structure of the data is roughly known. The benefit of BinX in this situation is providing transparency with respect to the original configuration of the data file and the system it is read on.

It is proposed that the programmer handle simple read cases in the following manner. After initialising the BinX runtime environment (this does things like figure out local byte orderings etc), the programmer creates a BxXmlFileReader object and uses it to parse the XML file, returning a pointer to a BxDataset object. The BxDataset object maintains references to the definitions of the data types in the file, and a reference to the file element or elements concerned (this is implemented as a collection of pointers to BxDataFile objects), within which the structure of the file is

defined. Allowance will be made for the fact that an arrayMultiFile element is allowed in place of a file element.

BxDatasetReader and BxDatasetWriter objects are how the programmer interacts with the dataset. The model is that the programmer asks the reader for data in a file referenced by the dataset, or that the writer is asked to write the data. Given that the programmer knows the overall structure of the data, it is up to him/her to make the appropriate calls to read/write the data. If the programmer tries to write an integer when the next data element is meant to be a float, the call will fail. The programmer reads the data element by element, asking the reader objects sequentially for their element values. The file element or file reader object keeps track of where in the file we are reading from. It is possible to poll the DatasetReader/Writer object to obtain the metadata about the next object to be read, without incrementing the file object's data pointer. Functionality will be provided to query the next element in the file about its type and name without reading its value.

Internally, the BxDatasetReader object utilises the services of BxDataFileReader objects. The role of these objects is to read raw data from the file. The raw data is passed back to the BxDatasetReader object, which checks with the dataset if any configuration transformations need to be carried out.

Consider the following simple example, without user-defined types:

```
<dataset>
  <definitions/><!- -empty - ->
<file src="data.bin">
  <integer-32 varName="code" comment=”some code for something”/>
  <arrayVariable varName="distances">
    <ieeeFloat-32/>
    <dimVariable>
  </arrayVariable>
  <ieeeFloat-32 varName=""/>
</file>

</dataset>
```

```
#include "BxXmlFileReader.h"
#include "BxSchemaElement.h"

// Program to read the data from the above BinX XML file and process in some way

int main()
{
  BxXmlFileReader*  xmlReader  =  new  BxXmlFileReader(“file.xml”)///  Reader  for  the
dataset
  BxDataSet*       ds;                                          /// provides data access


  // Initialise binx binary transformers for this platform
  // This sets local bit and byte ordering, and should be the
  // first line in any BinX program.
  BxRuntimeInfo::initialise();

  // parse the XML document and create the dataset
  ds = xmlReader->parse(xmlFile);
  // Set up a reader and initialise with a pointer to ds, for internal
  // reference when reading
  BxDatasetReader* reader = new BxDatasetReader(ds);
  reader->setFileByIndex(0);   // initialises to read from the first file in the
dataset
  int code;
  char* name=0;

  // Read the next integer value
  code = reader->nextDataValueInt();

  // alternatively we can check ahead for the next element
```

```
  if(reader->checkNextName("code") &&  reader->nextIsInteger32() )
  {
    code = reader->nextDataValueInt();
  }

  // now lets do the array. We read again thru the reader
  BxArrayValuesFloat* f;    // convenient object for holding returned data from array
read

  f = reader->newNextDataValuesFloatArray(); //returns null on fail, object contains
count

  // or, for memory
  reader->newNextDataValueFloatArray( f, 100 );// gets the next 100 elements
                                               // in the array and stores

  // or, between
  values= dfe->newNextDataValueFloatArrayBetween( &count, 0, 100 );
  process( values ); // does something with data
  delete [] values;

}
```

Example of read with defined datatypes

```
<dataset>
  <definitions>
    <typedef typeName="intensityData">
      <struct>
        <integer-32 varName="height"/>
        <integer-32 varName="length"/>
        <arrayFixed varName="intensity">
          <ieeeFloat-32/>
          <dim indexFrom="1" indexTo="100" varName="x"/>
          <dim indexFrom="1" indexTo="25"  varName="y"/>
        </arrayFixed>
      </struct>
    </typedef>
  </definitions>
  <file src="sss.bin">
    <integer-32 varName="count"/>
    <defType typeName="intensityData"/>
  </file>
</dataset>

// struct for intensityData
typedef struct s_intensityData{
    int height;
    int length;
    float xy[100][25];
  } intensityData_t;

// Program to access values in the data file described above
int main()
{
  BxXmlFileReader* xmlReader = new BxXmlFileReader("file.xml");
  BxDataSet*       ds;
  BxDatasetReader* reader;
  intensityData_t  data;

  // initialise binx binary transformers for this platform
  // sets local bit and byte ordering
  // first line in any BinX program
  BxRuntimeInfo::initialise();

  // Parse the BinX XML file to obtain the data information
  ds = xmlReader->parse(xmlFile);
  reader = new BxDatasetReader(ds);
  reader->setFileByIndex(0);

  // Get the first file element in the dataset
  // Read the integer
  int count = reader->nextDataValueInt();

  // Deal with the structure next.
```

```
  // This is tricky, especially when the struct contains arrays.
  IntensityData_t* data;
  data              =             (intensityData_t             *)             reader-
>newVoidNextValueDataStruct(sizeof(intensityData_t) );

}
```

Using BinX to write data

The following steps describe the basic BinX write mechanism.

1. The programmer builds a BxDataset object in memory by calls to the add methods of the object. This is the representation of the BinX XML file.
2. The programmer creates an instance of BxDatasetWriter and initialises it with a pointer to the dataset object.
3. The programmer initialises the writer to point at a particular file in the dataset.
4. Calls to the writer's write methods are made passing appropriate data.

Consider the following example, where data is output to a binary in the form described by the following BinX file.

```
<dataset>
  <definitions>
    <typedef typeName="pointType">
      <struct byteOrder=”bigEndian” bitOrder=”littleEndian”>
        <integer-32 varName="x"/>
        <integer-32 varName="y"/>
      </struct>
    </typedef>
  </definitions>
  <file name=”f.bin”>
    <Integer-32 varName="code"/>
    <arrayVariable varName="pointData">
      <defType typeName="pointType">
      <dimVariable>
    </arrayVariable>
  </file>
</dataset>

// this program writes an array of pointtypes to a file

int main()
{
  //initialise
  BxRuntimeInfo::initialise()

  BxTypedef* pointType = new BxTypedef(“pointType”);
  BxInteger32* bxIntX = new BxInteger32();
  bxIntX->setVarName(“x”);
  BxInteger32* bxIntY = new BxInteger32();
  bxIntY->setVarName(“y”);

  pointType->addElement(bxIntX);
  pointType->addElement(bxIntY);  // these must be done in the appropriate order
  bxIntX = bxIntY = NULL; // control passed to typedef

  // the next thing to do is set up the dataset object to contain these definitions
  // and the details of the structure of the file
  BxDataset* ds = new BxDataset();
  ds->addDefinition(pointType);

  // this is the representation of the integer in the file element
  BxInteger32* someInteger = new BxInteger32;
  someInteger->setVarName("code");

  // we start a file element to add to the dataset
  BxDataFile* dfe = new BxDataFile(“f.bin”);
  dfe->addField(someInteger);

  // we next add a description of the array of structs
  BxArrayVariableDefinedType* dataArray =
```

```
      new BxArrayVariableDefinedType( pointType );

  dfe->addField(dataArray);

  // add the new file description to the dataset.
  ds->addFile(dfe);
  ds->initForWriting();  // does anything necessary.

  // So far we have created the dataset
  // The next step is writing to the corresponding file
  // One approach is to build the data in memory bit by bit and write out using the ds
  // data for the example
  int count = 100;

  // We need:
  //    write mechanism for primitives
  //    write mechanism for arrays
  //    write mechanism for structs

  BxDatasetWriter* writer = new BxDatasetWriter(ds);
  if(!(writer->writeValueInteger32( count )))
  {
    // error
  }

  // We now demonstrate the procedure for writing structs and arrays.
  pointType[] points;  int length;

  if(!writer->writeArrayStructs(points, sizeof(pointType), length))
  {
    //error
  }
  delete ds; //etc.

}
```

Note: it is possible for the dataset to be initialised by reading from an XML file in the equivalent manner for reading.

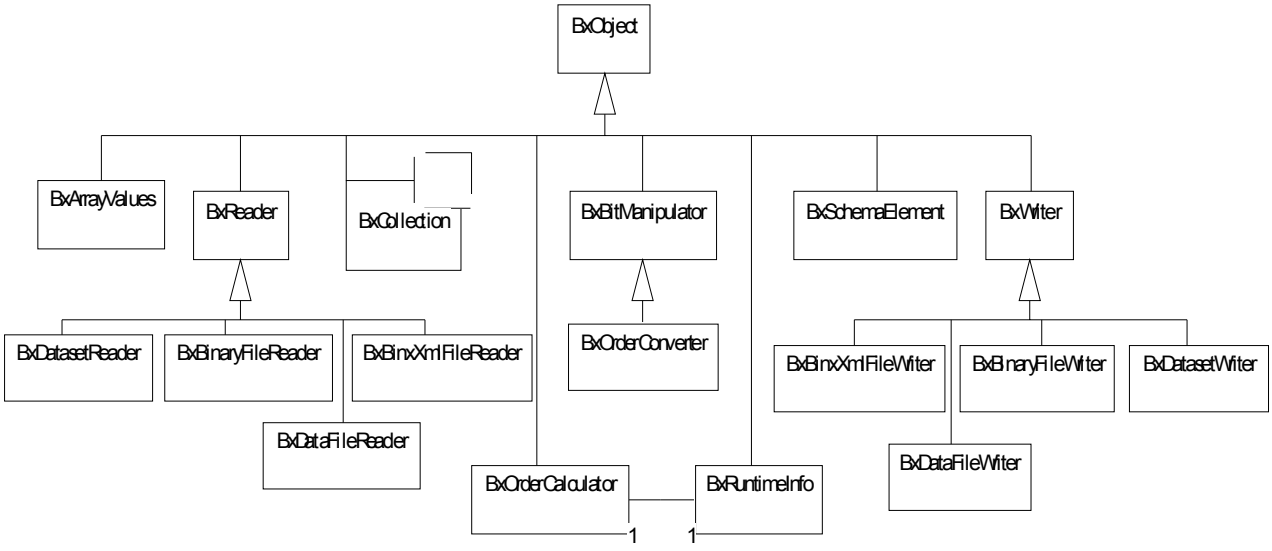
A note on using the library

This is a brief description of how the library might be used to build data access and manipulation tools. When performing structural or configuration changes on BinX data, we make use of the implied XML representation of the data. Every BinX data file could conceivably be expanded into an XML document. Often this will not be practical as the size of the XML data would be too large for memory. In situations like this it will be possible to use the functionality described above to sequentially read the data in a BinX data file, and output the data as a stream of XML elements. This stream can be interpreted as SAX/SAX2 events, and tools written to perform transformations or queries on these events.

General tools for browsing arbitrary data sets may or may not be provided. The functionality that we need in addition to that described in the preceding examples is that which provides metadata about the BinX elements. This is described below.

Implementation

The rough structure of the proposed class hierarchy is illustrated in the following diagram. See the accompanying UML model in BinX.mdl.



Class Design Overview

All classes are derived from the BxObject classes, which provides error logging, string handling and other useful functionality, as well as a general interface to all classes.

For consistency, all reader classes are derived from BxReader, and all writer classes are derived from BxWriter.

The BxCollection template class functions as a dynamic, ordered collection of pointers, and is used by a number of other classes.

Any element that can be present in a BinX XML file is derived from BxSchemaElement. An extended class hierarchy for these objects is presented below.

The remaining classes, BxOrderCalculator, BxOrderConverter and BxRuntimeInfo, provide binary transformation and information services.

Class Functionality

In this section we discuss the functionality of the class hierarchy and propose where various functions will be located. The functionality to be discussed will be as follows:

- BxSchemaElement reflection - on BxSchemaElement.
- Bx data type reflection – on BxTypedElement
- Access to data in binary file, converted – BxDatasetReader, BxDatasetWriter objects using BxBinaryFileReader/Writer objects.
- Access to raw data –BxBinaryFileReader/Writer
- Access to metadata – BxSchemaElement/BxTypedElement
- Runtime order information - BxRuntimeInfo
- Type size – individual types
- Array values read or to be written: BxArrayValues and descendents

In the following examples any method calls that are to be exposed to the user/programmer will be provided both as a conventional method call and a static

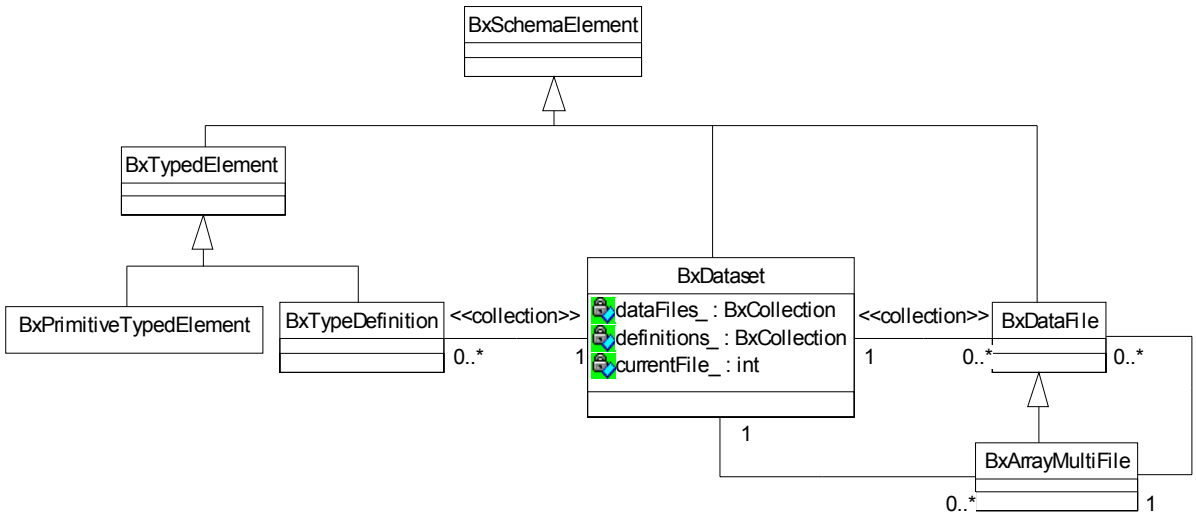
wrapper. A C interface will be provided in the form of appropriately modified global functions.

The classes embody three main areas of functionality. These are:

BinX Element representation	All classes derived from BxSchemaElement, including primitive and data types.
File Access and Manipulation	BxReader and BxWriter derivatives
Binary Order and Type conversion	BxOrderCalculator and associated classes.

BxSchemaElement/BxTypedElement derived classes

The class hierarchy for the objects which model the schema elements looks like this:



BxTypedElement

A typed element represents one of the allowed value types in BinX, be it primitive, defined type or array. An object derived from this type contains information about the configuration, size, variable-name and so on of a block of data in a BinX data file. Each BxTypedElement derived class corresponds directly to one of the BinX types. A BinX data file is considered to consist of an ordered collection of typed elements. Typed elements are so called to distinguish them from elements that do not represent data, such as the dataset element, or the file element.

It is necessary to provide a mechanism whereby the programmer can query an individual BinX typed element for its type. This will be the case when general-purpose tools are being provided. We will implement this at the level of BxTypedElement, and all subclasses override the appropriate method.

The size of the typed element is also stored here, as is its offset in the binary file it is stored in, if known. The size is set at compile time for the primitive types, but must be calculated at runtime for the data types, hence the virtual methods. The offset may also be unknown until the previous element has been read. Size is important when dealing with structs to ensure the correct amount of data has been read, or the correct amount of memory allocated etc. These should be implemented to log an error at this level.

```
// in BxCppTypes.h
enum BxTypedElementType

{
typeShort16,      typeUnsignedShort16,      typeInteger32,      typeUnsignedInteger32,
typeLongInteger64,  typeUnsignedLongInteger64,  typeIeeeFloat32,    typeIeeeDouble64,
typeIeeeQuadruple128,  typeBit1,  typeUnicodeCharacter32,  typeCharacter8,  typeVoid0,
typeEnum32,   typeArrayStreamed,   typeArrayVariable,   typeArrayFixed,   typeStruct,
typeUnion
}  //extend if necessary.

// BxTypedElement.cpp
class BxTypedElement:public BxSchemaElement{

private:
  char*                       varName_;  // as only Typed elements can have variable
names.
  BxTypedElementType        type_;
  int                      offsetInBytes_; // only meaningful here
  int                      sizeInBytes_;
  bool                      hasBeenRead_;  // has the data represeneted by this object
been read before? Will be useful in extensions
  bool                     hasBeenWritten_;
  BxConfiguration*         configuration;  // Used by BxDatasetReader
// indicates what order transformations are required, calculated on parse
public:
// constructor/destructor
  BxTypedElement();
  ~BxTypedElement();
// accessing
  virtual BxTypedElementType getDataType();
  char*        getVarName();       // name of the variable
  virtual void setDataType(BxTypedElementType typeToSet);
  virtual int  getSizeInBytes();
  virtual void setSizeInBytes();   // subclassed e.g. by arrays etc
  virtual int  getOffsetInBytes();
  virtual void setOffsetInBytes(); // needs to be calculated or read by object
  void              setBitOrder(BxBitOrder bitOrder); // sets val of bitOrder_ in
configuration
  BxOrder      getBitOrder();

  // methods to return the value of a particular type
  static BxTypedElementType getTypeShort16(){return typeShort16; };
  static BxTypedElementType getTypeUnsignedShort16();  //etc
  static BxTypedElementType getTypeInteger32();
  static BxTypedElementType getTypeUnsignedInteger32();
  static BxTypedElementType getTypeLongInteger64();
  static BxTypedElementType getTypeUnsignedLongInteger64();
  static BxTypedElementType getTypeIeeeFloat32();
  static BxTypedElementType getTypeIeeeDouble64();
  static BxTypedElementType getTypeIeeeQuadruple128();
  static BxTypedElementType getTypeBit1();
  static BxTypedElementType getTypeUnicodeCharacter32();
  static BxTypedElementType getTypeCharacter8();
  static BxTypedElementType getTypeVoid0();
  static BxTypedElementType getTypeEnum32();
  static BxTypedElementType getTypeArray();
  static BxTypedElementType getTypeArrayStreamed();
  static BxTypedElementType getTypeArrayVariable();
  static BxTypedElementType getTypeArrayFixed();
  static BxTypedElementType getTypeStruct();
  static BxTypedElementType getTypeUnion();
}
```

Configuration details

Every element in the BinX file has a certain configuration, i.e. bit and byte order and blocksize combination. This may differ from element to element throughout the file. These attributes are reified into the BxConfiguration class, which looks like the following.

```
class BxConfiguration
{
```

```
private:
  BxBitOrder  bitOrder_;
  BxByteOrder byteOrder_;
  int         blockSize_;
public:
// getter and setter for each of these attributes, constructor, destructor.
}
```

Every BxSchemaElement contains an instance of this class, and child elements in the XML file inherit the parent's configuration, unless specified. In most cases, bit and byte order will be uniform throughout the file.

Access to metadata

Presented at the BxSchemaElement level and overridden by subclasses. Note that some of the metadata does not apply to all elements. Nevertheless we present the interface at this level. The constructor should set these all to null/blank.

```
private:
  BxByteOrder byteOrder_;
  BxBitOrder  bitOrder_;
  int         blockSize_;
  bool        ignore_;
  char*       comment_;
  char*       info_;


//Metadata Access
// N.B. for calls like blocksize we need to return the correct blocksize
// wherever (in the BinX file) that has been specified.
// public
  BxBitOrder  getBitOrder();  // returns bit order
  BxByteOrder getByteOrder(); // returns byte order
  int         getBlockSize(); // blocksize
  bool        getIgnore();    // ignore field (e.g. padding)
  char*       getComment();   // return user comment string
  char*       getInfo();      // return info field
  // etc.
```

BxDataset

The BxDataset object contains all of the information about the dataset that is needed to successfully read or write binary data. This means it contains information about the structure of the files and the defined types involved. The developer asks this object for a reference to the relevant file element, from which they then read, or to which they then write.

```
private:
  BxCollection<BxTypedef*>          typeDefs_;        // info on the defined types
  BxCollection<BxDataFile* > fileElements_;    // info on binary files
public:
  // accessing
  BxDataFile* getFileElement(int index) const; // returns pointer to file element
  void           addFileElement(BxDataFile* dfe);
  BxTypedef      getTypedef(int index) const;
  void           addTypeDef(BxTypedef* td);
```

A note on memory management for this object: during the creation of this object, BxTypedef and BxDataFile objects will be created and added to this object, be it by the programmer in the case of a write, or a call to BxXmlFileReader::parse() in the case of a read. Responsibility for these created objects then falls to the BxDataset object, and the descructor must make sure to delete them.

BxDataFile

This element type represents a data file element from the schema and contains details of the structure of the data file within it. Whenever a BxDatasetReader object

read call is made, reference is made to a BxDataFile object to see if the appropriate read is being carried out. The same applies when writing is carried out.

```
private:
  BxCollection<BxTypedElement *> types_;    // contains ordered collection of types in
file
  int                       currentIndex_;    // points to current element, updated on
read
                                              // or write.
  BxDataFile             dataFile_;
// methods private
  void incrementCurrentIndex();
  void decrementCurrentIndex();
public:
  // accessing
  void  setDataFileName(char* fileName);
  char* newGetDataFileName();
  // binary data access simple API
  // all of these methods check the collection types_ to see if the next data in
  // the file is in fact an int, double etc
```

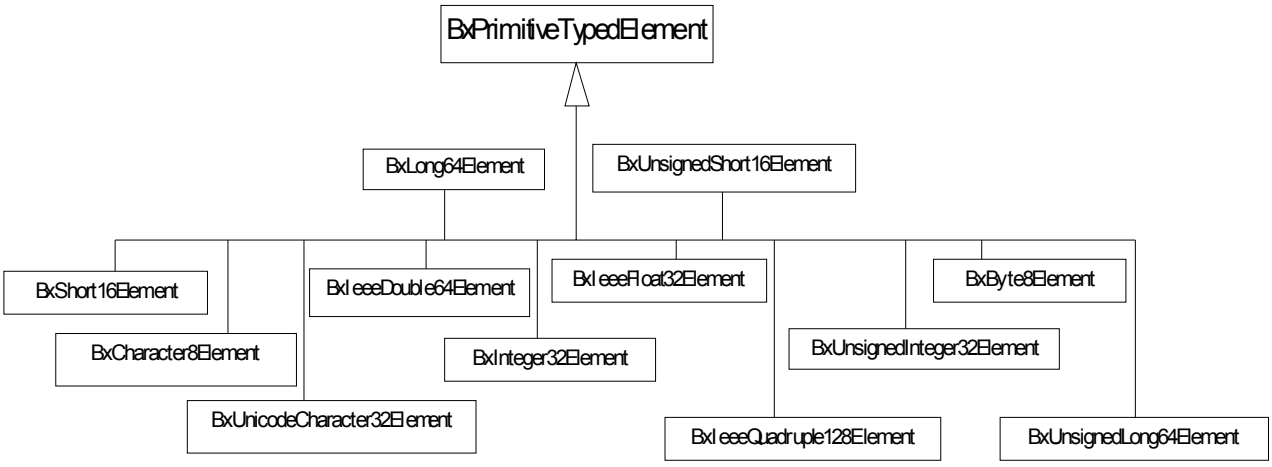
BxTypedef

This is a class that describes the elements that are contained by a struct or union. It looks like:

```
class BxTypedef : public BxObject
{
  public:
    addElement (BxTypedElement type);
    BxTypeElement* getElementByIndex(int index);
  private:
    char* defTypeName;
    BxCollection<BxTypedElement* >;  // collection of elements within the type
   // note any of these elements may also be a defined type.
   // have add, nextElement, elementAt etc methods

}
```

Primitive type implementations



The above are all directly derived from BxPrimitiveTypedElement, which derives from BxTypedElement, and implement the functionality defined there. As stated previously, these store details about size, metadata and offset for each of the elements in the file. An example of implementation for BxInteger32 is given.

```
class BxInteger32Element : public BxTypedElement
```

```
{
// this inherits the following member variables from BxTypedElement:
// type_, offsetInFile_, sizeInFile_, configuration_;

private:

public:

}
BxInteger32Element::getSizeInFile()
{
  return sizeof(BxInteger32Value);
}
BxInteger32Element::BxInteger32Element(bitOrder, byteOrder, blockSize)
{
  setBitOrder (bitOrder);
  setByteOrder(byteOrder);
  setBlockSize(blockSize);
  sizeInFile = 4;  // bytes
// etc
}
```

Arrays

Arrays may contain primitive types or defined types. Arrays have one or more dimensions, and can be spread across one or more files. The basic array class contains a variable number of dimension objects. The size of the array can generally be determined by multiplying the element size by the number of elements, which can be determined from the dimensions.

The library is expressly designed **not** to provide array access by element reference. It is merely designed to provide configuration independent access to the data in the files. It is thus proposed that in the case of multi dimensional arrays, the programmer accesses the array as if it were one dimensional, and then performs whatever processing necessary to extract the points required. This fits in with the performance requirements of the library.

The array classes all inherit from the class BxArray. This contains a collection of BxDimension objects, which represent each dimension of the array. BxArray looks like:

```
class BxArray : public BxDataType
{
private:
  BxCollection <BxDim* > dimensions_;


public:
  int calculateSize();
}

// other array classes

class BxArrayElementShort16;
class BxArrayElementUnsignedShort16;
class BxArrayElementInteger32;
class BxArrayElementUnsignedInteger32;
class BxArrayElementLongInteger64;
class BxArrayElementUnsignedLong64;
class BxArrayElementDouble64;  // etc

class BxArrayElementTypedef;
```

Array values

In order to pass the values stored in an array back from a read function, or to a write function, we specify a number of container classes. The root class here is

BxArrayValues. All of these classes contain a long integer, which is the length of the data. They also contain a BxTypedElementType variable, which is set in the constructor and represents the type of the primitive elements. The subclasses contain pointers to data of the appropriate type. These classes will be instantiated with an integer, and space for the data will be allocated on instantiation. This is mainly a set of convenience classes used to shuttle data from place to place. This could be implemented as a template class.

Useful Binary/Data Functionality

C++ type mapping

We need to accurately define the mapping between each of the BinX primitive types and the corresponding C++ type. This is done in BxCppType.h using a number of typedefs.

```
typedef BxShort16Value                    short;
typedef BxUnsignedShort16Value            unsigned short;
typedef BxInteger32 Value                 int;
typedef BxUnsignedInteger32Value          unsigned int;
typedef BxLongInteger64 Value             long int;
typedef BxUnsignedLongInteger64Value      unsigned long int;
typedef BxIeeeFloat32Value                float;
typedef BxIeeeDouble64Value               double;
typedef BxIeeeQuadruple128Value           long double;
typedef BxBxUnicodeCharacter32Value       int;
typedef BxBit1Value                       bool; // ??
typedef BxCharacter8Value                 char;
//typedef BxVoid0                      ??
//typedef BxEnum32                     ??
```

BxOrder

This also appears in BxCppTypes.h and typedefs BxByteOrder and BxBitOrder as ints. This defines the symbolic constants BIG_ENDIAN and LITTLE_ENDIAN.

```
#define BIG_ENDIAN     1
#define LITTLE_ENDIAN  0
typedef BxByteOrder    int;
typedef BxBitOrder     int;
```

BxOrderCalculator

The BxOrderCalculator provides the functionality to check they byte and bit ordering of the local file-system and internal memory byte and bit orders. These are calculated once at the start of the program and stored in static member variables belonging to BxRuntimeInfo. Note local memory bit order is always big-endian.

```
#include "bxorder.h"

private:

public:

  BxByteOrder calcLocalMemoryByteOrder();
  BxByteOrder calcLocalFileByteOrder();
  BxBitOrder  calcLocalFileBitOrder();
```

BxRuntimeInfo

This class contains a collection of static variables holding the bit and byte order on the local system. These are calculated by calling BxRuntimeInfo::initialise(); itself

static. These static member functions are called by reader/writer functions to see what transformations need to be carried out.

BxOrderConverter

This class takes data and converts the bit and byte ordering as required. The data is passed using a pointer, so the transformed data can be stored in the original memory location. The calls are implemented as static to avoid having to instantiate transformer objects. This example demonstrates the API for double types. Overloaded methods will be provided for every BinX C++ type.

```
private:

public:
  static void transformBitOrder(BxDouble *toTrans);
  static void transformByteOrder(BxDouble *toTrans);
  static void transformArrayBitOrder(BxDouble* toTrans, int length);
  static void transformArrayByteOrder(BxDouble* toTrans, int length);
  static void transformArrayBitByteOrder(BxDouble* toTrans, int length);
```

BinX Data Access

The class diagrams for file access can be seen in the main class hierarchy diagram above.

BxDatasetReader, BxDatasetWriter

The APIs for reading and writing BinX binary data as described in the BinX XML file is located on these objects. The objects are initialised with a reference to a BxDataset object and make reference to it during the read/write process. The API will look like the following.

```
  // 1) writing

  int         nextValueInt();    // returns the next integer value in the file
  unsigned int nextValueUnsignedInt();

  double      nextValueDouble();
  long double nextValueQuadruple();
  long int    nextValueLongInt();
  unsigned long int nextValueUnsignedLong();
  short       nextValueShort();
  unsigned short nextValueUnsignedShort();
  BxByte      nextValueByte();
  char        nextValueChar();
              nextValueUnicodeCharacter();

  // corresponding array calls
  // these do the type checking ie to see if the next array is one of ints etc
  int*     newNextValuesArrayTo(int to, int* count);
  int*     newNextValuesArrayAll(int* count);
  int*     newNextValuesArrayBetween(int from, int to, int* count);
  … …
  void*    newNextValuesArrayStruct(… sizeof(struct));
  // reading structs
  void* newNextReadStruct(int size); // pass sizeof(struct) and cast

  // 2) writing
  // all return success or not
  bool writeInteger(int toWrite);
  bool writeDouble(double toWrite);
  bool writeChar(char toWrite);
  bool writeLongInt(long int toWrite);
  bool writeUnsignedInt(unsigned int toWrite);
  bool writeUnsignedShort(unsigned short toWrite);
//etc.
  bool writeIntegerArray(int* toWrite, int numElements);
//etc.
```

Note that checking to see whether the appropriate method has been called is not carried out here; rather, these objects call the methods of an instance of BxDataFileReader/Writer which checks the details of the referenced BxDataFile.

BxXmlFileReader

This object handles the BinX Xml file that describes the BinX data file. This class provides the functionality to parse the Xml file and build the dataset object. An outline class structure is:

```
class BxXmlFileReader
{
private:

public:
 BxDataSet* parse();
 BxDataSet* parse(char *binxXmlFile);
 void       validate(char* fileToValidate);
 void       validate(char* fileToValidate, float schemaVersion);
}
```

Use cases are as above. The validate methods should provide the means to validate an arbitrary XML file against the BinX schema, perhaps against different versions of the schema. The DOM Parser is used to extract the information necessary to build the representation.

BxXmlFileWriter

This file provides functionality to write an in memory dataset representation to a physical file. This will be implemented using the DOM API, as this provides a convenient means of building XML trees and writing them out. The following illustrates its use.

```
// writing
BxDataSet ds;
// build up ds as normal, see above for API

BxXmlFileWriter* writer = new BxXmlFileWriter();
writer->writeXmlFile(ds, "filename.xml");   // writes binx xml file to filename.xml
```

BxDataFileReader, BxDataFileWriter

These provide functionality equivalent to the BxDatasetReader/Writer classes for the BxDataFile object. They are utilised by BxDatasetReader/Writer. For the read, the BxDatasetReader instantiates a BxDataFileReader for every data file in the data set. These objects are initialised in the following manner:

```
BxDataFileReader* dfr = new BxDataFileReader(BxDataFile* dataFile);
```

Each of these objects provides an equivalent for each of the BxDatasetReader/Writer API methods. These methods use a BxBinaryFileReader to read raw bytes from the file. They then check the dataFile description for configuration information, and **perform the appropriate configuration transformations** before passing the data back to the caller in native representation.

BxBinaryFileReader

Unfinished!!!
This class provides raw, unconverted read access to bytes in the binary file. This is the level where blocksize is taken into account, and the file pointer for each dataFile is stored here also.

The methods provided will depend on a single method, readBytesRaw(). A seek or padding method will also be provided to cope with blocksize.

```
public:
  // reader
  BxInteger32 readInteger();
  bool        readIntegerArray();
  // etc
private:
  FILE * fp;
```

BxBinaryFileWriter

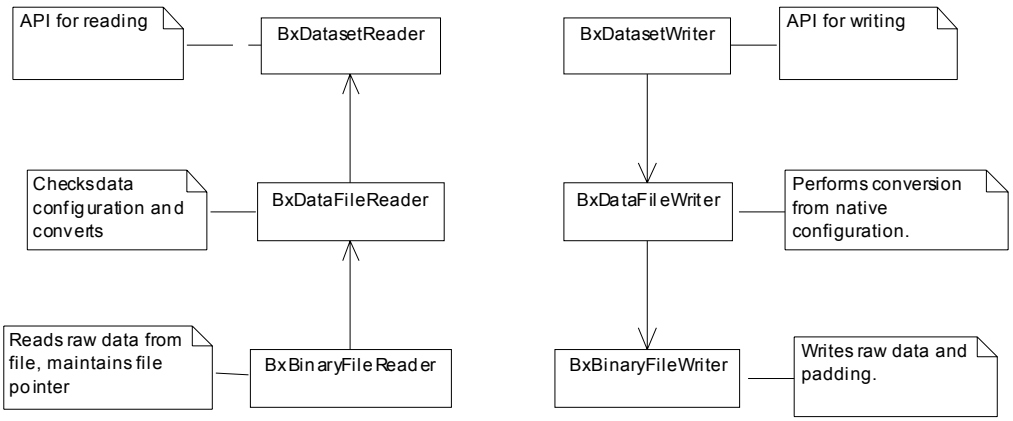
This class provides an interface to write data to a binary file. The methods of this class are passed data with internal memory ordering and convert it to the correct output ordering. For checking, the methods should return the number of bytes they have written.

Other readers/writers

It is possible that we may wish to provide a random access read and write API. It is proposed that this functionality be provided by BxRandomAccessDatasetReader/Writer objects.

Summary of Data Access

The following diagram contains a summary of the read and write methods in terms of the above objects.



General Functionality

There will be a top level class BxObject to provide general functionality such as logging, error functions, string comparison and manipulation functions etc.

The template class BxCollection provides dynamic array functionality for any type.

C/Fortran interfaces

We need to use the objects described above to create a set of C functions and header files for use in C and possibly Fortran programs.

Implementation Plan

It is proposed that the implementation proceed along the following steps.

1. Build the library to deal with one file that contains primitive types only.
2. Modify to deal with arrays of simple types.
3. Modify to deal with multiple files.
4. Modify to deal with simple structures, i.e. small ones that do not contain other structs or arrays.
5. Design a mechanism/API for dealing with heavily nested or large structs.
6. Deal with unions.

In order to achieve goal 1 above, we should follow this implementation path:

1. Build the order transformation and calculation and runtime objects: LARGELY COMPLETE (2 days left)
2. Build BxSchemaElement, BxTypedElement, BxPrimitiveTypedElement and the primitive types (4 days)
3. Build the raw binary Read/Write objects (BxBinaryFileReader/BxBinaryFileWriter) accommodating for blocksize (5 Days each = 10 days)
4. Build the BxDataset object using the BxTypedElements already built to test it (3 days)
5. Build the BxXmlFileReader/Writer objects(5 days each for initial version)
6. Build the BxDatasetReader/BxDatasetWriter objects (7.5 days each)

Component dependencies: the SchemaElement and derivatives are independent of the order objects and can thus be developed in parallel. The dataset object can be built once at least some of the element objects are built, or if a TestElement object is built from the correct inheritance tree. The XmlFileReader/Writer objects can be built once the dataset object is built. The binary reader and writer objects can be built when we have at least one or two of the element objects built for testing (they need the metadata).

Proposed path for two developers:

1. Halve the development of the SchemaElement objects and PrimitiveType derivatives.
2. One developer develops the BinaryFileReader/Writer objects. The other builds the Dataset object and the XMLFileReader/Writer object
3. One developer builds the BxDatasetReader and the other the writer.

Following a parallel path like this gives a development time of the order of 20 days.

In order to achieve step 2 of the overall development, the following steps need to be carried out.

1. Code the array value container objects (8 days)
2. Code the array element objects (8 days)
3. Modify the reader/writer objects to deal with arrays (12 days)

Step 3 requires the following.

1. Code the struct element objects
2. Modify the reader and writer objects to deal with structs.

Directory structure

The directory structure in the CVS repository is arranged as follows. The code is in the gridserve CVS repository. The module is src/WP5/Binx-LibCpp. In this module are subdirectories src, containing the C++ files, documents containing project

documentation (including this one) and projects, which contains projects for Visual Studio, Makefiles for Unix and any other build configuration. The subdirectory src contains a further subdirectory called testbinx, which contains classes to run unit tests on the BinX library classes.

Testing the BinX library

The BinX library needs to be tested on a number of levels. Of primary importance are unit testing of the library class methods, functionality testing using generated and real test data, and stress/performance testing to ensure we reach the performance goals. It is also necessary to consider sources of test data. Platform independence of written data is also necessary. We need be able to write data on one system and successfully read it on another, as this is obviously a central requirement of the library!

A note on testing this design: the design is under weekly technical review. The people involved are generally happy with the state of the design and the evolution it has been undergoing. It is hoped that this process will identify and correct any design errors that reduce the degree of detail, intersection or merit of the design. Naturally any faults discovered during implementation will be fed back into this document.

Test data

There are two main sources of test data: generated data used for unit/functional testing, and real user data for functionality and performance testing.

We need a means of generating interim test data. It is proposed that some of the code used in the Jaja demonstrator be modified for this purpose. This can be used to output binary files according to accompany hand-written BinX XML files. Test data should be provided for each stage of the development. Functionality borrowed from the LEDataStream classes utilised by BinX can be utilised to provide little-endian data.

Unit Tests

The unit test

Functionality

sd

Performance testing

asdf

Appendix – coding standards

C++ Coding Style Guidelines For BinX

GC 20/11/00

This document outlines the C++ coding style decided upon by the BinX team. Following a set of guidelines will bring the following benefits.

1. Allow developers to switch between different sections of code without being thrown by different styles (or no style at all).
2. Make the code appear much more professional when examined by third parties (who may have financial influence).

1. Should allow the generation of automatic source documentation.

Files General

- Header files have a .h suffix eg. BxObject.h.
- Code files have a .cpp suffix eg. BxObject.cpp
- Files take their name from the class eg. BxSchemaElement.h and BxSchemaElement.cpp for class BxSchemaElement.

Header Files

- No method bodies in the header files unless inlined.
- No inlined methods unless absolutely necessary for speed.
- Only one class per header file (and hence per code file).
- Use #ifndef in header files to prevent multiple inclusions. Use BinX_[Class] for the name of the variable being defined eg.

```
#ifndef BinX_BxSchemaElement
#define BinX_BxSchemaElement
... class definition
#endif
```

Naming

- All class names are prefixed with 'Bx' to prevent problems when integrating with external code.
- All class names are a concatenation of one or more words, each with a leading capital. eg. BxSchemaElement, BxTypedElement.
- Instance variables are a concatenation of one or more words (where the second and subsequent words are capitalised), with a trailing underscore eg. type_, value_, dataSet_.
- Class (static) variables follow the same rules as for class names (apart from the 'Bx' prefix), but have a trailing underscore eg. Value_.
- Local (temporary) variables follow the same rules as for instance variables but without the trailing underscore eg. name, value.
- Method names follow the same rules as for local variables eg. cancel(), schedule(...), isArray(), handle(), logClassName().
- Acronyms in names are handled by capitalising the first letter of the acronym only eg. BxWriterXml instead of BxWriterXML.

Auto Documentation

Doxygen has been chosen as the tool for producing automatic documentation from our C++ files. See http://www.stack.nl/~dimitri/doxygen for details. It has been chosen in preference to doc++ as it can produce output in a variety of formats, and

allows instance variables to be commented on the same line, helping to keep the source code readable.

The basic idea is that code comments are formatted in a special Doxygen way, allowing the tool to pick them up and produce pretty HTML source code descriptions and class hierarchy diagrams etc.

The following policy is recommended for deciding what should have a Doxygen comment.

1. All API methods in .cpp files should have a Doxygen comment.
2. Method declarations in .h files need NOT have a Doxygen comment (pointless duplication).

There are various options for the format of a Doxygen comment. My preferred one is /// (C++ style with an extra slash).

For documenting variables on the same line you can use ///<. The < is supposed to mean that the comment is being associated with the thing to the left.

You can use HTML tags such as <P> to specify a new paragraph within a large comment eg.

```
/// This is the end of a long paragraph.
///
/// <P> And this is the start of another one.
```

To run doxygen from a PC:

1. Make sure you are logged in as gridserve eg. by typing \\sambahost\gridserve into a file browser and entering the appropriate password when it asks you.
2. Copy ~gridserve/tools/doxygen/doxy.bat file to the directory where your source code lives.
3. Execute the doxy.bat file. This will generate a doc subdirectory in your current directory and fill it with lots of auto documentation.

Comments

- Comments are meant to be in English and so include normal punctuation such as full stops.
- All header files should have a comment at the top explaining what the class does eg. for BxSchemaElement.h:

```
// ################################################################
// BinX
// $Id$
//
/// Superclass of all elements in a BinX file
// ################################################################
```

- Note that the comment should also include the string *Id*. On check-in to CVS, this string is automatically expanded to contain versioning information about the file.
- All variables declared in header files should have a comment explaining briefly what they represent eg.

```
  BxBinaryFileReader* client_;  ///< Reader for raw data
```

A method whose name doesn't fully reveal its purpose should have a short comment in the header file, preferably on the same line for readability eg.

```
  // Diagnostics.
  void         logClassName() const; // Print out the object's class and address.
  virtual void logVars()      const; // Print info about current variable values.
  void         logObject()    const; // Print out class name and variables.
  void         logObjectl()   const; // As logObject() but with newline.
```

All methods which have anything more than an utterly obvious single-line body should have a comment in the code file. At the minimum it should describe what the method is trying to achieve, what it should return, and any side-effects or assumptions. eg.

```
void BxOpTimer::expire()
/// Called by simulation engine.  The timer has 'gone off'.
/// SIDE EFFECT: Wakes up the client.
{
  ...
}
```

Note that this comment has been put below the method header line. This helps prevent comments being left behind when cutting and pasting. Unfortunately, for Doxygen to work the comment must come before the opening {.

Protocols

To ease location, methods are grouped into protocols. The first protocol is 'Initialising.', containing constructors and destructors etc.. The second is 'Accessing.', containing basic access methods for the class (eg. get and set for key variables). Subsequent protocols are added in alphabetical order.

The class definition is split into public, protected and private sections (in that order), so it may not be possible to adhere to alphabetical order for protocols. For example, instance variables usually (always if you ask me!) go in the protected or private sections in an 'Instance Data' protocol eg.

```
protected:
  // Instance Data.
  OTclManager* manager_;       ///< Pointer to the sole manager instance.
  int          numInstances_;  ///< Number of instances of this class.
  char*        metaName_;      ///< The name of the class's class (!).
```

In header files, method declarations go on consecutive lines, with a single blank line before protocol headings eg.

```
  // Accessing.
  virtual char* className() const;

  // Packet Handling.
  void send   (BxPacket* p, int outputIndex); // For communication to
  void receive(BxPacket* p, int inputIndex);  // simulation engine.
```

In code files, method definitions are separated by a single blank line, with two blank lines before protocol headings eg.

```
  // Accessing.
```

```
  char* BxOrderCalculator::className() const {
    return " BxOrderCalculator";
  }


  // Packet Handling.

  void BxExtOutputPort::send(BxPacket* p)
  /// Subclased to send this packet out to the simulation engine.
  {
    extModel_->send(p, index_);
  }
```

Regardless of protocol order, the order of method bodies in the code file should be the same as the order of method declarations in the header file (again to assist in finding methods).

Layout General

- Tabs are 2 spaces wide (prevents 'staircase' appearance and leaves space for comments at the end of a line).
- Alignment is used, where helpful, to improve readability. This may mean aligning equals signs or variable declarations within methods, or comments in header files eg.

```
BxDelay::BxDelay(double delayTime) {
  delayTime_  = delayTime;
  packet_     = NULL;
  timer_      = sim()->factory()->newTimer(this);
}
```

- Method bodies should ideally contain less than about 12 lines of code. although up to an emacs page in length is okay. There should be a good reason for any method longer than that! Dividing algorithms into smaller chunks often reduces errors and allows reuse of lower-level sections in other algorithms.
- The * for pointers should be attached to the type not the variable name eg. *char* str* not *char *str*. This is because the * alters the type of the variable and so is better associated with the type definition.
- When #including other files, the standard ones (eg. stdio) go first in angle brackets, followed by the others in quotes. It's helpful to say why something's been included eg. in Object.cpp:

```
#include <stdio.h>    // For printf.
#include <string.h>   // For strlen, strcpy.
#include <stdarg.h>   // For multiple arguments like printf.
#include "BxObject.h"
```

Use of Const Keyword

The keyword *const* should be used wherever possible to increase the safety of the code. Next to a method declaration it means "this method will not modify any of the object's variables". Next to an argument it means "this method will not modify the argument in any way". Example of both:

```
BxParameter* paramNamed(const char* name) const;
```

Use of Virtual Keyword

When 'overriding' or 'subclassing' methods, they should be declared virtual. Technically it is sufficient to declare only the top-level version (ie. the one highest up the class hierarchy) as virtual. However, to make it obvious which methods are ripe for overriding, all versions should have the virtual keyword too. Eg. In BxObject:

```
  // Accessing.
  virtual char* className() const;   // Each subclass should override.
```

Then again in BxBlackBox:

```
  virtual char* className() const;
```

Ultimate Superclass

All new classes are made descendents of BxObject in order to inherit various generic functionality as described below.

Messages for Developers

- Diagnostic messages should use the inherited *logf* not *printf*. Doing so will allow such messages to be easily turned off in a customer build.
- Other diagnostic methods are inherited from BxObject eg. logClassName(), logVars(), logObject() etc.
- Note that all new classes should override the method className() as this is called by some of the diagnostic methods.

Messages for Users

These should be logged with inherited methods userError(...), userWarn(...), or userInfo(...), depending on the severity of the condition. Using these methods will ensure consistency in the information displayed to the user.

Example Header File

```
// ###########################################################################
// BinX
// $Id$
//
/// Class representing a delay primitive which holds a packet for a
/// specified time and then sends it on.  Any new packets arriving while
/// a packet is being held are dropped.
// ###########################################################################

#ifndef BinX_BxDelay
#define BinX_BxDelay

#include "BxPrimitive.h"
#include "BxTBxer.h"


class BxDelay : public BxPrimitive {
 public:
  // Initialising.
  BxDelay(double delay);
  ~BxDelay();

  // Accessing.
  virtual char* className() const;

  // Packet Handling.
  virtual void receiveFrom(BxPacket* p, BxPort* port);
```

```
  // Timer Handling.
  virtual void wakeUp(BxTimer* timer);  // Respond to timer by sending packet.

 private:
  // Instance Data.
  BxTimer*  timer_;       ///< Timer for implementing the delay.
  BxPacket* packet_;      ///< The current packet being delayed.
  double    delayTime_;   ///< Time by which packets are delayed.
};

#endif
```

Example Code File

```
// ##################################################################
// BinX-DiffServ
// ##################################################################

#include "BxDelay.h"


// Initialising.

BxDelay::BxDelay(double delayTime) {
  delayTime_  = delayTime;
  packet_     = NULL;
  timer_      = sim()->factory()->newTimer(this);
}

BxDelay::~BxDelay() {
  delete timer_;
}


// Accessing.

char* BxDelay::className() const {
  return "BxDelay";
}


// Packet Handling.

void BxDelay::receiveFrom(BxPacket* p, BxPort* port)
/// Delay the argument packet, or drop it if we've got one already.
{
  if (timer_->isPending()) {
    // Sending us packets faster than we can cope with them.
    // Drop the new packet.
    delete p;
    logf("BxDelay dropped packet\n");
    return;
  }
  packet_ = p;
  timer_->schedule(delayTime_);
}


// Timer Handling.

void BxDelay::wakeUp(BxTimer* timer)
/// The timer has gone off so send the packet on.
{
  if (timer_ != timer) {
    userError("BxDelay::wakeUp", "woken up by an external timer");
  }
  send(packet_);
  packet_ = NULL;
}
```

Policy for JAVA Code

To be done…