

edikt

BinX 1.2

Developer's Guide

edikt::BinX Developer's Guide

edikt::BinX 1.2 Developer's Guide
Edition 1.3
© University of Edinburgh 2005

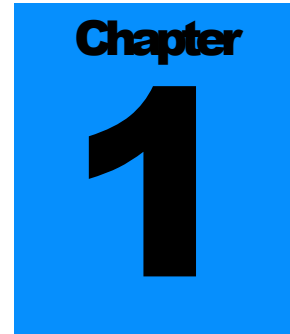
edikt

Old College, University of Edinburgh
Edinburgh, Scotland, EH8 9YL
www.edikt.org support@edikt.org

Table of Contents

1	Preface	4
1.1	The purpose of this manual	4
1.2	How to use this manual	4
1.3	Who should read this manual	4
1.4	Notations used in this manual	4
1.5	Related documentation	5
2	An introduction to BinX	6
2.1	Prerequisite concepts	6
2.2	BinX concepts	7
3	Describing binary data using the BinX language	10
3.1	An overview of the BinX language	10
3.2	The XML schema and document elements	11
3.3	The primitive data type elements	12
3.3.1	Primitive data types supported	13
3.3.2	String data type	13
3.4	The complex data type elements	14
3.4.1	Structures	14
3.4.2	Arrays	15
3.4.3	Unions	19
3.5	The type definition elements	20
3.5.1	Defining data types	20
3.5.2	Referencing user-defined types	21
4	Processing binary data using the BinX API	23
4.1	Programming environment	23
4.1.1	API versus internal structure	23
4.2	BinX class inheritance structure	23
4.2.1	Abstract superclasses	24
4.2.2	File-related classes	24
4.2.3	Simple data types	25
4.2.4	Complex data types	25
4.2.5	Helper classes	26
4.3	An overview of the BinX API	26
4.4	Reading a binary file	27
4.4.1	Opening a BinX document and its associated binary file for input	27
4.4.2	Reading sequentially from a simple binary file	28
4.4.3	Reading from a simple binary file in random order	29
4.4.4	Accessing primitive data types	30
4.4.5	Accessing String data type	31
4.4.6	Accessing complex data types	32
4.4.7	Access to binary file meta-data	36

4.5	Runtime errors and warnings	38
4.6	Creating a binary file	39
4.6.1	Creating a structure of data objects in memory	39
4.6.2	Creating a new binary file with an existing BinX document	43
4.6.3	Creating a new associated BinX document	43
Appendix A	BinX Language Reference	46
A.1	Language Elements	46
Appendix B	External API	52
B.1	Primitive data type protocol	52
B.2	Complex data type protocol	56
Appendix C	Sample Programs	57
C.1	The <code>BinXConverter</code> program	57
C.2	The <code>DataBinx</code> program	57
C.3	The <code>DataBinxParser</code> program	57
C.4	The <code>GenSchemaBinx</code> program	57
C.5	The <code>MergeArray</code> program	58
C.6	The <code>ParseBinx</code> program	58
C.7	The <code>ReadHeader</code> program	58
Appendix D	DataBinX Utilities	59
D.1	The <code>GenDataBinx</code> utility	59
D.2	The <code>DataBinxParser</code> utility	59
Appendix E	Common mistakes	60
Appendix F	Error messages	62



1 Preface

This introductory chapter:

- Identifies this manual's purpose and audience.
- Explains how to use the manual and its structure.
- Explains the naming and highlighting conventions used throughout the manual.
- Lists related documentation.

1.1 The purpose of this manual

To explain the capability of the BinX language and library to developers

1.2 How to use this manual

Chapter 1 is a general introduction to BinX. It identifies prerequisite concepts with which the reader should be familiar, and then gives an overview of the BinX language and the BinX library.

Chapter 2 describes in more detail the BinX language which is used to describe a binary data file

Chapter 3 describes in more detail writing a program to read or write a binary data file using the BinX library of functions.

1.3 Who should read this manual

The following groups should find this manual of interest:

- Application and computer scientists and IT professionals who are interested in using the BinX library.
- Data administrators who will develop BinX documents describing binary data.
- Programmers and software developers who will develop programs that use the BinX library to access binary data.

1.4 Notations used in this manual

The following conventions are used in this manual:

- The main body of the text in this manual is written in a proportional font as follows:
Normal text
-

- Examples of both program code and BinX language documents are written in a fixed pitch font with a lightly shaded background as follows:

```
<dataset><integer-32 varName="bank_balance" /></dataset>
```

- Within both program code and BinX language documents, text which needs to have values substituted to make the whole program or document complete is identified with an italicised fixed pitch font with a lightly shaded background as follows:

```
<dataset><integer-32 varName="variable name" /></dataset>
```

- Within both program code and BinX language documents, certain text may be highlighted for emphasis. This is done to draw the reader's attention to a particular part of the program text or language document. For example, the previous example would perhaps have been clearer as:

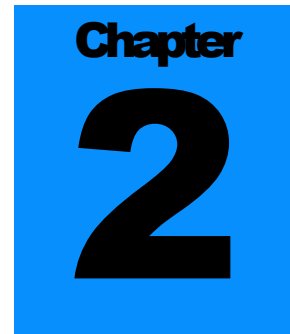
```
<dataset><integer-32 varName="variable name" /></dataset>
```

- There are many references in the text to member functions of classes in the C++ programming language. Such references do not generally cite the complete signature of the member function, but merely identify the member function name and often the class name. Thus the text may contain `setByteOrder()` or `BxDataObject::setByteOrder()`, but not `virtual void BxDataObject::setByteOrder(BxByteOrder)`. In particular, unless a member function parameter is to be explicitly referred to in the text, it is generally not included. The full signatures of methods are given in the 'BinX API Reference' manual.

1.5 Related documentation

The reader may need to consult the rest of the BinX documentation to find out more about BinX. The following manuals contain further BinX documentation:

- BinX Installation Guide and Reference
 - BinX API Reference
-



2 An introduction to BinX

BinX is a language with which to describe the layout of binary data files.

In this section we describe:

- The main prerequisite concepts on top of which BinX is built. The reader will need to have a good understanding of these before reading any further in this manual, as they are not explained any further here. References are given to other material on these prerequisite concepts which the reader may consult for further explanation.
- The main concepts of BinX itself. The purpose of this section is to explain these concepts to the reader who does not already have a good understanding of them. An outline is given of each of these concepts, and how they relate to each other. The intention is that this should be sufficient to enable the reader to use more detailed material on BinX in the rest of the manual.

2.1 Prerequisite concepts

BinX is built on the following concepts and technologies:

- XML
XML – eXtensible Markup Language – is a meta language with which to specify individual markup languages (like BinX). Further information about XML and related technologies can be found at <http://www.w3.org/xml/>. Individual documents whose structure is marked up with XML elements (tags) are known as XML documents.
 - XML schema
An XML schema is an XML document that defines additional markup elements (tags) which can be used in (other) XML documents. XML schema is one of two ways to define XML tags (the other is using a Document Type Definition (DTD), which is not itself an XML document).
(If all this seems too recursive, then just reflect that XML schema is itself defined in an XML schema!)
Further information about XML Schema can be found at <http://www.w3.org/XML/Schema>.
 - DOM and SAX - programmatic access to XML
The XML standards cover only the contents of files marked up with XML tags. To access an XML document file from a program, means either parsing the tag structure in your own code or using one of two standard APIs to invoke parsers to do it for you. These APIs are known as DOM (Document Object Model) and SAX
-

(Simple API for XML). The difference between the two is that DOM parses a whole document into a complete structure of in-memory objects, whereas SAX parses the document into a stream of XML elements which are presented sequentially to the application program. (Note that although it is not visible to the application programmer, the BinX library uses a SAX interface to parse a BinX document) The DOM can be further explored at <http://www.w3.org/DOM/>, SAX can be followed up at <http://www.saxproject.org/>.

- Xerces
Xerces is an open source product of the Apache foundation XML project. The current release of the BinX library uses the Xerces C++ XML SAX parser. More information about Xerces can be found at <http://xml.apache.org/#xerces>.
- Library
The term library (short for program library) is used in this document to describe the standard method of packaging executable code for reuse in an Open Systems (or Windows) environment. For details on precisely the kinds of library technology supported in a given operating system, see the reference documentation supplied with the operating system or in the case of Linux at <http://www.dwheeler.com/program-library/>.
Note that the BinX library must be statically linked with a BinX application program.

2.2 BinX concepts

- Binary data
We define binary data as any data which has its meaning encoded according to some (usually very economical) scheme. Unless the encoding scheme used happens to be one of the standard text representation schemes (ASCII, Unicode, EBCDIC, etc), such data cannot be displayed by a standard text editor. Indeed the data cannot be understood by any other program which does not contain details of the particular encoding scheme used.
The fundamental purpose of BinX is to simplify programmatic reading and other manipulation of binary data, firstly by representing the encoding scheme for a binary file with an associated BinX Language document, and secondly by providing a library of functions which a program can invoke to decode the binary data in the file automatically, allowing the program to access a binary data field in terms of the associated BinX document rather than having to calculate addresses and possibly perform data type conversions within the program.
In principle there is no reason why BinX should not support any and every binary data encoding scheme, (including, of course, ASCII, Unicode and eBCDIC) but in the current release only certain well-established standard representation formats, mostly for numbers, are supported. These include various integer and floating point formats, and also bytes and characters.
- The BinX language
The BinX language is an application-specific markup language used to describe binary data files. (It is a dialect of XML, and is therefore itself defined syntactically in an XML schema.) The BinX language codifies some meta-data¹ about binary data files,

¹ **Meta-data** Data *about* data.

Meta-data is definitional data that provides information about, or documentation of, other data managed within an application or environment.

the elements (tags) in the language identify the binary file, its data structure elements, and also the individual fields in the file.

The motivation for the BinX language is to give some of the virtues of XML tag language representation (relatively easy for humans to understand and make sense of) without losing the compactness of binary data.

Experiments suggest that a fully decoded and tagged XML representation of a complex binary file could take as much as four times the space of the binary original, partly because all binary data would need be rendered into a textual representation, and partly because of the need for textual markup to denote each individual occurrence of a data element or structure.

Because the BinX language description of a binary file is separate from the original rather than being embedded within it, it is possible to avoid this potentially enormous overhead. Many common or repeated elements can be defined once rather than repeatedly. Furthermore the original binary data can remain unchanged in the binary file, supporting any existing data access required.

- BinX language document

A BinX language document is an XML document which describes a given binary file using the BinX language. A BinX language document consists of a sequence of BinX language (XML) elements which describe the individual binary fields in the binary file, plus other BinX language elements which describe any more complex data structure in the binary file. The following BinX document fragment describing part of a hypothetical bank account record gives a feel for what is involved:

```
<character-8 varName="account-type"/>
<string varName="account-name"/>
<short-16 varName="account-number"/>
<integer-32 varName="account-available-balance"/>
<integer-32 varName="account-uncleared-balance"/>
<short-16 varName="account-interest-rate"/>
```

Typically a BinX developer would write such a document to describe each binary file of interest. The BinX language is described in more detail below in Part 3, “Describing binary data using the BinX language”, and is defined below in Appendix A, “BinX Language Reference”.

Where contextually clear, the phrase ‘BinX language document’ is often abbreviated below to ‘BinX document’.

- BinX language schema

Because the BinX language is itself defined in a more abstract markup meta-language (XML), there is a more abstract document, the BinX XML schema (identified by <http://www.edikt.org/binx/2003/06/binx>) which formally defines the syntax of the BinX language itself. In this document each element which can appear in a BinX language document is defined using standard XML Schema elements such as `<element />`, `<complexType />`, `<attribute />`, etc.

For example, meta-data would document data about data elements or attributes, (name, size, data type, etc) and data about records or data structures (length, fields, columns, etc) and data about data (where it is located, how it is associated, ownership, etc.). Meta-data may include descriptive information about the context, quality and condition, or characteristics of the data.

(From the free on-line dictionary of computing <http://foldoc.doc.ic.ac.uk/>)

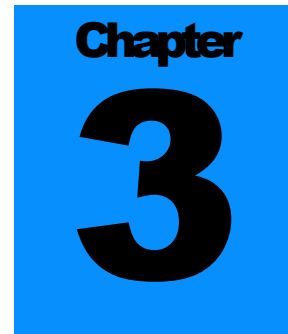
Typically a BinX developer would not explicitly need to read the BinX language schema at all, unless to clarify ambiguities in this document. However as with any other XML schema, the BinX language schema can, and should, be used to validate the XML syntax of a BinX language document (particularly if hand-coded rather than machine generated) using a standard XML tool. A list of such tools can be found at <http://www.w3.org/xml/Schema>.

- BinX library and API

The BinX library is a set of reusable subroutines which can be called from an application program to access the data in a binary data file. The subroutines use the BinX language description of the structure and content of the binary data file to enable the application program to access the data in the binary file without having to know its low-level layout.

Essentially this works as follows:

- The BinX language document is read and an in-storage model of the BinX language meta-data is built.
This defines the structure and content of the binary data file
 - The API then allows the application program to use the BinX library routines to retrieve binary file data in terms of the BinX language meta-data model. This includes facilities for navigating sequentially through complex data structures such as arrays so that (potentially) binary data files which are much greater in size than available memory can be processed.
-



Chapter 3

3 Describing binary data using the BinX language

In this chapter we introduce the elements of the BinX language (an application-specific markup language used to describe binary data files). We use an incremental approach to both the language elements, and the binary data file they describe. We start by showing how to use a basic group of language elements to describe simply-structured binary data files, and then progressively introduce more language elements and use them to describe progressively more complex binary data files. First, however, before getting into the detail of individual language elements we give an overview of the whole language, how the elements are grouped, and how the groups relate to one another.

3.1 An overview of the BinX language

The BinX language contains several groups of language elements:

1. Schema elements.

These elements identify that this is a BinX document (i.e. which conforms to the BinX schema).

Strictly speaking the first element is not part of the BinX language at all, but of the XML meta-language, however it is included here because it is required in every BinX document:

- `<?xml?>`
- `<binx>`

2. Document element.

This describes the BinX document itself rather than individual elements of content. This comprises the following element:

- `<dataset>`

3. Primitive data type elements.

These are used to describe the individual fields of the binary data file. This is by far the most numerous group, containing the following elements:

- `<byte-8>`
 - `<character-8>`
 - `<short-16>`
 - `<integer-32>`
 - `<long-64>`
 - `<unsignedByte-8>`
 - `<unsignedShort-16>`
 - `<unsignedInteger-32>`
 - `<unsignedLong-64>`
 - `<float-32>`
 - `<double-64>`
-

- `<string>`
4. Complex data type elements.
These are used to describe the data structure of the binary data file. This group contains the following elements:
 - `<arrayFixed>`
 - `<arrayVariable>`
 - `<struct>`
 - `<union>`
 - `<dim>`
 - `<case>`
 - `<sizeRef>`
 5. Type definition elements.
These are macros, used in an analogous manner to the typedef facility in the C or C++ programming languages, to assign a document-specific name to a data structure. The name can then be referenced repeatedly in the rest of the document. This group contains the following elements:
 - `<definitions>`
 - `<defineType>`
 - `<useType>`

The element groups above are given in order of increasing document complexity:

1. Even the simplest document describing an empty file has to contain elements from the first two (Schema and Document) groups.
2. A document describing the very simplest meaningful file (one fixed record) must contain elements from the first three (Schema, Document and Primitive data type) groups.
3. A document describing a file with more complex data structure needs to contain elements from the first four (Schema, Document, Primitive data type and Complex data type) groups.
4. A document describing a file with multiple occurrences of the same data structure may contain elements from all the groups.

The rest of this chapter is structured according to this incremental approach to both BinX document elements and BinX document complexity. Successive sections describe in more detail the document elements in each group, and also show how to use the elements to build documents to describe binary files of progressively greater complexity.

3.2 The XML schema and document elements

Every BinX document must contain the following sequence of elements:

- An XML declaration which identifies that it is an XML document, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

(Readers wishing to understand the meaning of this element further are referred to the XML specification at <http://www.w3.org/TR/REC-xml>.)

- A `<binx>` element which identifies that the XML document is in fact a BinX document (i.e. it uses the BinX schema as a namespace to define further valid elements that can be used in this document). This is achieved by the following:

```
<binx xmlns="http://www.edikt.org/binx/2003/06/binx"></binx>
```

The `<binx>` element in turn can contain two further elements:

- An optional `<definitions>` element which is used to define the user-defined type names outlined above. These are described in more detail below in section 3.5, "The type definition elements".
- A mandatory `<dataset>` element which describes the binary file:
 - The `<dataset>` element contains BinX language elements whose data structure reflects the data structure of the binary file described. These are the subject of many succeeding subsections of this document.
 - The mandatory `src` attribute specifies the name of the (single) binary file described. The data file is always assumed to be in the same directory as the BinX document; hence no path qualifier is necessary in the `src` attribute in order to identify a file. This also means that neither the schema nor the data file can be stored remotely.

This structure is illustrated in the following (vacuous) example, which describes an empty binary file:

```
<?xml version="1.0" encoding="UTF-8"?>
<binx xmlns="http://www.edikt.org/binx/2003/06/binx">
  <dataset src="UK.bin">
  </dataset>
</binx>
```

It is also possible to specify the endian-ness of the fields in the binary file using the `byteOrder` attribute of the `<binx>` or `<dataset>` elements. This specifies the endian-ness of the byte order of the individual elements within the file. It defaults to the endian-ness of the operating system (either "littleEndian" or "bigEndian").

If the byte order is specified, all the individual data elements contained within the file inherit the attribute value unless they themselves have a `byteOrder` attribute value specified.

3.3 The primitive data type elements

The primitive data type elements are used to describe the individual binary fields in a binary file. They correspond closely to the supported encoding schemes used for binary data, e.g. bytes, characters, integers and floating point numbers. The actual tags, however, are specific to each individual field format, e.g. `<byte-8>`, `<unsignedInteger-32>`. A complete list of tags and the primitive data types they describe is included at the end of this section.

All primitive data type tags may have the `varName` attribute specified. This gives the field a name with which it can be referenced when using the BinX API.

All multi-byte primitive data types may have the `byteOrder` attribute specified. The value specified for a primitive data type will always override the value specified for any containing element in the BinX document.

The following example illustrates the use of a sequence of primitive data types to describe a simple binary data file consisting of a single record. (The record concerned describes various aspects of the UK)

```
<?xml version="1.0" encoding="UTF-8"?>
<binx xmlns="http://www.edikt.org/binx/2003/06/binx">
  <dataset src="UK.bin" byteOrder="littleEndian">
    <character-8 varName="CurrencySymbol"/>
    <short-16 varName="VatRate"/>
    <short-16 varName="StandardIncomeTaxRate"
      byteOrder="bigEndian"/>
    <integer-32 varName="GDP"/>
    <float-32 varName="landAreaInAcres"/>
    <float-32 varName="annualRainfallInInches"/>
    <float-32 varName="averageDailySunshineInHours"/>
  </dataset>
</binx>
```

3.3.1 Primitive data types supported

The following table illustrates the BinX tags which describe a range of primitive data types.

Data Type	BinX Tag
8-bit (signed) byte	<byte-8>
8-bit unsigned byte	<unsignedByte-8>
Character (signed)	<character-8>
16-bit short integer	<short-16>
Unsigned 16-bit short integer	<unsignedShort-16>
32-bit integer	<integer-32>
Unsigned 32-bit integer	<unsignedInteger-32>
64-bit long integer	<long-64>
Unsigned 64-bit long integer	<unsignedLong-64>
IEEE single precision floating point number	<float-32>
IEEE double precision floating point number	<double-64>
String ASCII (unsigned)	<string>
The empty or void data type (an artificial data type used in complex data structures)	<void-0>

BinX Tags for Primitive Data Types

3.3.2 String data type

There are three types of string that can be specified. A fixed length string, a variable length string (the length of the string given by a data value within the binary data, just before the string) and a delimited string (a string terminated with a specified ASCII character). The following example illustrates the three types:

```
<dataset src="strings.bin">
  <string size="16" varName="first"/>
  <string sizeType="byte-8" varName="varstr1"/>
  <string sizeType="short-16" varName="varstr2"/>
  <string delim="\n" varName="delim1"/>
  <string delim="#255" varName="delim2"/>
  <string delim="&gt;" varName="delim3"/>
  <string varName="default"/>
```

```
</dataset>
```

The length of the fixed length string is the number of bytes specified in the schema (in the example string “first” is 16 bytes long). The length of the variable length string is given by a data value of the specified type in the schema (in the example the length of string “varstr1” is the data value of the byte just before the string and the length of string “varstr2” is the data value of the short just before the string).

In the example there are three delimited strings. The ASCII character that terminates the string can be specified in three ways:

- By an escaped character.
- By a number between 0 – 255 preceded by the ‘#’ character representing an ASCII.
- By a XML escaped character representing an ASCII.

The first string “delim1” is terminated by an escaped character; the second string “delim2” is terminated by a number representing an ASCII character and the third string “delim3” is terminated by a XML escaped character representing the ASCII character ‘>’. The default string is a delimited string terminated by a NULL character.

3.4 The complex data type elements

In previous sections we have seen how to describe the binary encoding of very simple binary data files with BinX language elements. In this section we describe how to deal with the more complex data structures that are more commonly encountered in real binary files. Specifically we describe three structural elements which aggregate component elements in different ways. These can then be combined in a variety of mutually recursive ways to deal with a wide variety of common data structures found in binary files. These elements are:

1. Records (structs) – finite sequences of different component elements.
2. Arrays (of different types) – ordered collections of components of identical type
3. Unions – discriminated sets of alternate components.

Like the primitive data types, all of these elements may have the `varName` attribute specified. This gives the field a name with which it can be referenced when using the BinX API.

Similarly all may have the `byteOrder` attribute specified. If a byte order is specified, all data elements contained within that element inherit the attribute value unless they themselves have a `byteOrder` attribute value specified.

3.4.1 Structures

A structure (a record, or finite sequence of component elements), is defined using the `<struct>` tag.

A `<struct>` contains a list of component elements. These can include both primitive data types and other complex data types, arrays, unions, or even another `<struct>`.

A simple example of a typical BinX `<struct>` can be seen below:

```
<struct>
  <byte-8 varName="x"/>
  <short-16 varName="y"/>
  <float-32 varName="f"/>
```

```
</struct>
```

The `<struct>` tag may have the `varName` attribute specified. This gives the `<struct>` a name with which it can be referenced when using the BinX API.

In addition, the `<struct>` tag may also have the `dim` attribute specified. This is used to specify a single dimension to the structure, so that all the elements within the `<struct>` are repeated the number of times specified by the `dim` attribute. This attribute only applies to the `<struct>` tag. For example, in the example below, the structure actually contains 50 occurrences of the pair of integer and float data elements:

```
<struct dim="50">
  <integer-32/>
  <float-32/>
</struct>
```

Examples of more complex uses of the `<struct>` tag, emphasising the mutually recursive nature of the complex data type tag composition rules are given in succeeding sections.

Note that the `<struct>` and `<dataset>` tags are very similar. The `<dataset>` tag, described in the previous section, is essentially a special kind of `<struct>` tag used to identify the top level of the data structure in a binary file. Because of this it has a special `src` attribute, as already mentioned, that identifies the name of the binary file described.

3.4.2 Arrays

3.4.2.1 Array concepts

- **Array**
An array is a collection of element instances of identical type, ordered according to the sequence in which the element instances are stored in the binary file. The type of the instance may be any of the fixed-length primitive or complex data types.
- **Array index**
Individual element instances within an array are identified by a (zero-based) index. BinX supports multi-dimensional arrays, where the index is a vector of non-negative integers rather than just a single scalar non-negative integer. Indexes are defined in the BinX language with `<dim>` tags, for example:

```
<dim name="monthIndex" indexTo="11"/>
```

could be used to index a set of experimental results for each month in a given year (the month varying from 0 to 11).

Multi-dimensional index vectors are defined using compound `<dim>` elements where one `<dim>` element is contained in (at least) one other `<dim>` element. For example:

```
<dim name="yearIndex" indexTo="99" >
  <dim name="monthIndex" indexTo="11"/>
</dim>
```

could be used to index a larger set of results for a whole century (the year varying from 0 to 99).

These examples also highlight that:

- `<dim>` has a `name` attribute, analogous to the `varName` attribute of the explicit data types.
- Dimension value ranges are specified using the `indexTo` attribute. In a normal statically-determined dimension the `indexTo` attribute is mandatory, producing C-style zero-based array dimensions. (The concept of a statically-determined dimension is covered below under “Different Array Types”.)
- Outermost dimension

In a multi-dimensional index, the outermost dimension of the index vector is that which increments most slowly as the element instances are scanned in the order they are stored in the binary file. Thus the outermost component element of a `<dim>` element defines the outermost dimension, so that the outermost dimension in the example immediately above is named `yearIndex`.

3.4.2.2 Different array types

BinX supports two different types of arrays, differing according to when the outermost index dimension is determined. (In both array types, all the inner index dimensions are determined when the BinX document describing the file is written.)

- Fixed-length array

In a fixed length array the number of elements in the outer dimension is determined when the BinX document describing the file is written, like those of the inner dimensions.
- Variable-length array

In a variable length array the number of elements in the outer dimension is determined at the time of binary file creation and is stored as a binary value of integer type immediately preceding the sequence of element instances in the array.

3.4.2.3 Fixed-length arrays

Fixed length arrays are described using the `<arrayFixed>` element. The `<arrayFixed>` element must contain exactly one (component) element of any data type (defining the type of the collected instances) and a `<dim>` element defining the dimensionality of the index.

Because a fixed-length array has a statically-determined outermost index dimension, the `<dim>` element should be a nested structure of `<dim>` elements with a `<dim>` element for each index dimension.

An example of a typical fixed-length BinX array can be seen below:

```
<arrayFixed>
  <integer-32 varName="hourlyMeasurement"/>
  <dim indexTo="6" Name="dayOfWeek">
    <dim indexTo="23" Name="hourOfDay"/>
  </dim>
</arrayFixed>
```

In this example, the component data element contained within the array is of type `integer-32`. The array has two dimensions, the slower moving dimension being 7 elements long, the faster moving dimension being 24 elements long. Note that because indexes are zero-based, the actual number of elements in each dimension is the attribute value plus one.

3.4.2.4 Variable-length arrays

Variable-length arrays are described using the `<arrayVariable>` element. Like the `<arrayFixed>` element described above, the `<arrayVariable>` element must contain exactly one (component) element of any data type defining the type of the collected instances. However the way that the index dimensionality is specified is different.

Recall that in a variable-length array the size of the outermost index dimension is stored in the binary file as a binary field of integer type immediately preceding the collection of element instances. This is reflected in the BinX language - this size field is described with a `<sizeRef>` element of integer type, which must be the first component element of the `<arrayVariable>` element, i.e. immediately before the element defining the collected instances' type. In addition, although the `<arrayVariable>` element contains a `<dim>` element to describe the index dimensionality of the array, the outermost `<dim>` element may not have the `indexTo` attribute specified (because it is determined by the value of the field in the file described by the `sizeRef` element).

An example of a variable-length array can be seen below:

```
<arrayVariable>
  <sizeRef>
    <short-16 varName="cardinalityCount"/>
  </sizeRef>
  <double-64 varName="arrayData"/>
  <dim name="outerDim">
    <dim name="listOfDoubles" indexTo="127"/>
  </dim>
</arrayVariable>
```

The example above defines a variable-length array of double precision floating point numbers. The array has two dimensions, the inner dimension specifies a sequence of 128 double precision floating point numbers, the number of elements in the slower moving outer dimension is not known at document creation time, but is stored in a short integer at the start of the array.

3.4.2.5 Array or structure?

Both the `<array>` and `<struct>` tags can be used to describe collections of binary data elements.

Furthermore, collections can be described

- by nesting both arrays and structures to multiple levels
- by having both arrays and structures in the hierarchy (elements of an array can be a structure, and elements of a structure can be an array).

A simple example of this is shown below:

```
<arrayFixed>
  <arrayFixed>
    <float-32 varName="data values"/>
    <dim indexTo="9"/>
  </arrayFixed>
  <dim indexTo="4"/>
</arrayFixed>
```

In this example, the component data element contained within the outer array is another fixed-length array. The inner array contains 10 elements of type `float-32`; the outer array contains 5 elements of the inner fixed-length array. (But note that, for reasons set

out below in this section, it is recommended to use a single two-dimensional array to achieve the same end.)

This section contains some guidance to assist the data administrator in choosing the appropriate tag to use.

In some cases it is clear that only one of the tags can be used. For example:

- An array cannot be composed of variable length instances. It is therefore not possible to define an array of any of the following types:
 - variable length string
 - delimited string
 - variable length array
 - union

In such a case it is only possible to use a structure to describe the collection.

The following example shows how to define a collection of variable length arrays:

```
<struct dim="3">
  <arrayVariable>
    <sizeRef><byte-8/></sizeRef>
    <integer-32/>
    <dim/>
  </arrayVariable>
</struct>
```

The BinX document above contains three variable arrays, each array containing a different number of elements (as given by the `sizeRef` element).

- The number of elements in a structure must be determined when the document is defined. This means that if the number of elements in a collection is defined in the binary file then it is only possible to use a variable length array.

In many other cases, however, there is more than one valid way to describe a given collection of binary elements. For example, any collection that can be described by a fixed array can also be described by a structure. (In the case of a multi-dimensional fixed array, the structure would actually be a nested set of structures.)

However, when making the choice, the data administrator should also bear in mind the following implementation considerations.

- Differences between the two tags:
 - The `<struct>` tag is implemented as:
 - an object representing the structure as a whole
 - a linked list of component objects, one per component

When the `dim` attribute is specified, the number of objects in the linked list is a multiple of the `dim` attribute.

- The `<array>` tag is implemented as:
 - an object representing the array as a whole
 - a single prototype object of the same type as the array elements

The prototype object is used as a 'cursor' to access individual array elements.

This means that the memory requirements for a large collection of binary data elements could be considerably less when defined as an array, than when defined as a structure. It is therefore recommended that in general an administrator should use arrays to describe binary element collections in preference to structures.

- Inefficiency of nested arrays
The implementation of `<array>` was not designed with recursion in mind. It is therefore recommended that the administrator should use a multi-dimensional array to describe a binary element collection in preference to a nested set of arrays. (Any collection that can be described with nested arrays can also be described with a single multi-dimensional array.)

3.4.3 Unions

3.4.3.1 Union concepts

- Union
A union is an element that is made up of a number of alternative (base) elements. The value of any instance of the union in the binary file is an instance of one and only one of the base data elements.
A typical use for this is in describing records which may be of different types. For example, a file of legal entities, containing two types of records, one type describing individuals, and another type describing organisations, might be described using BinX as an array of elements each of which is the union of an individual structure and an organisation structure.
- Discriminant
Each instance of a union type in the binary file may, as described above, be of any of the given base elements. The discriminant for the union is a field, present in each instance of the union in the binary file, which identifies which base element is present in that instance.
In a BinX union the discriminant must be a single field of integer type, an instance of which occurs at the start of each instance of the union element.
- Base data elements
The base data elements can be of any of the primitive or complex data types.

3.4.3.2 Union declaration

A union declaration must contain the following sequence of elements:

- A `<discriminant>` element which contains the data element defining the discriminant field for the union.
The discriminant field can only be of one of the integer-based data types, namely, byte-8, short-16, integer-32, long-64 and the four equivalent unsigned integer types.
- A series of `<case>` elements each of which:
 - specifies through the `discriminantValue` attribute the value of the discriminant which identifies this case.
 - contains an element specifying the base element of the union for this case.

An example of a typical BinX `<union>` can be seen below:

```
<union>
  <discriminant>
    <short-16/>
  </discriminant>
```

```

<case discriminantValue="32">
  <float-32/>
</case>
<case discriminantValue="64">
  <float-64/>
</case>
<case discriminantValue="0">
  <void-0/>
</case>
</union>

```

The above example defines a union with a short integer to discriminate between the following data:

- if the value of the discriminant read from the binary file is 32, then the following data is a single precision floating point number;
- if the value of the discriminant read from the binary file is 64, then the following data is a double precision floating point;
- if the value of the discriminant read from the binary file is 0, then there is no following data. This case uses the empty data type `<void-0>` to denote the absence of data.

3.5 The type definition elements

The BinX language offers a facility to define user-defined data types which can then be referenced in the rest of the document. This is achieved using a macro-like facility, analogous to the `typedef` facility of the C and C++ programming languages.

The actual “macros” defining the user data types are included (where present) in the `<definitions>` component element of the `<binx>` element of a BinX document. This is an optional component of the `<binx>` element. An example of the `<definitions>` element is shown in the following document which describes the same empty binary file as in the example in section 3.2, “The XML schema and document elements” above:

```

<?xml version="1.0" encoding="UTF-8"?>
<binx xmlns="http://www.edikt.org/binx/2003/06/binx">
  <definitions>
  </definitions>
  <dataset src="UK.bin">
  </dataset>
</binx>

```

The type definition elements used to define user types in the `<definitions>` element are described below in the next section.

Referencing those definitions is described below that in the immediately subsequent section.

It should be noted that the main purpose for the user-defined data types is to simplify the structure and readability of a BinX document. It is particularly useful for documents in which the same structural sub-element is incorporated into the document in more than one place, typically in the dataset element of the document.

3.5.1 Defining data types

The `<definitions>` element referred to in the previous section consists of a sequence of `<defineType>` elements, each defining a user type for later reference.

Each `<defineType>` element has a `typeName` attribute which specifies the user-defined type name.

Each `<defineType>` element also contains a single component element which must be of complex data type (i.e. a structure, union or array). This is an anonymous element which serves to define the underlying type of the user-defined type, it must not have the `varName` attribute specified.

The effect of the `<defineType>` element is that a reference in the rest of the document to the value of the `typeName` attribute is replaced by an element of the complex component type.

The following example illustrates the use of user-defined types to define a Pascal-style string as an array of `<character-8>` elements:

```
<defineType typeName="PascalString">
  <arrayVariable>
    <sizeRef><unsignedByte-8/></sizeRef>
    <character-8/>
    <dim/>
  </arrayVariable>
</defineType>
```

Note that the `byteOrder` attribute must not be specified in `<defineType>` elements.

3.5.2 Referencing user-defined types

A user-defined type is referenced in the `<dataset>` component of the document using the `<useType>` element.

The `<useType>` element establishes the user-defined type reference using the `typeName` attribute. This should have the same value as the `typeName` attribute specified in the `<typeDef>` element referenced.

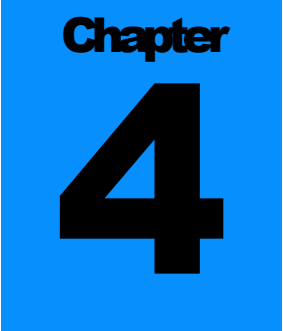
The following example combines elements of the previous two examples to show a binary data file with a defined type and a simple record layout comprising a Pascal-style string:

```
<?xml version="1.0" encoding="UTF-8"?>
<binx xmlns="http://www.edikt.org/binx/2003/06/binx">
  <definitions>
    <defineType typeName="PascalString">
      <arrayVariable>
        <sizeRef><unsignedByte-8/></sizeRef>
        <character-8/>
        <dim/>
      </arrayVariable>
    </defineType>
  </definitions>
  <dataset src="UK.bin">
    <useType typeName="PascalString" varName="name"/>
  </dataset>
</binx>
```

In addition to its use in the dataset element of a document, a user-defined type can also be referenced in another user-defined type definition. In the following example the `PascalString` type (as defined above) is used in a further type definition.

```
<defineType typeName="PascalString">
  <arrayVariable>
```

```
<sizeRef><unsignedByte-8/></sizeRef>
<character-8/>
<dim/>
</arrayVariable>
</defineType>
<defineType typeName="MyType">
  <struct>
    <integer-32 varName="ID"/>
    <useType typeName="PascalString" varName="Name"/>
    <double-64 varName="Value"/>
  </struct>
</defineType>
```



Chapter 4

4 Processing binary data using the BinX API

In this chapter we describe how to process a binary file using the BinX API.

The first two sections are groundwork for the main body of the exposition. In the first section we describe the programming environment that is required, in the second we give an overview of the class inheritance hierarchy.

Succeeding sections then describe the BinX API in terms of processing increasingly complex documents in increasingly demanding ways.

4.1 Programming environment

In version 1.1 of BinX, application programs to use the API must be written in the C++ programming language and compiled, and linked with the BinX library using the GCC compiler. More details on how to do this are documented in the BinX Installation Guide.

4.1.1 API versus internal structure

The BinX application programmer needs to be aware of two different but related things:

- The BinX API (Application Programming Interface). This is the interface exposed by the BinX library to be used by the application programmer. It is the documented and supported way to use the library.
- The BinX library modular structure. This is the detailed structure of the BinX library code, in terms of classes, member functions, etc. This structure can be seen in the C++ header files distributed along with the binaries.

The BinX API is essentially a subset of the full modularity of the library.

Those parts of the modular structure which are not included in the API are for internal use only and should not be used by application programmers. Use of the internal use only functions will not be supported. Many of the internal use only member functions which are not included in the BinX API are neither appropriate nor safe for other than internal use. Others are not included because they may be re-factored in future releases, or are not fully developed or tested.

4.2 BinX class inheritance structure

In this section we give a brief outline of some of the inheritance structure of the BinX classes. More complete documentation can be found in the BinX API Reference Manual.

(Note that in the class inheritance diagrams that follow, the use of class names in parentheses denotes that the primary description of that class is in another subsection of this section.)

4.2.1 Abstract superclasses

```
BXObject
  BxDataObject
```

Currently almost all BinX classes are subclasses of a single abstract superclass, `BXObject`. `BXObject` implements common behaviour with regard to error handling, logging, etc. In addition, classes representing data types have a common abstract superclass `BxDataObject`. This class implements behaviour common to all fields and structures in the binary file (e.g. size and position related behaviour).

4.2.2 File-related classes

The classes which are responsible for file-related behaviour are shown below:

```
(BXObject)
  BxBinxFile
    BxBinxFileReader
    BxBinxFileWriter
  BxBinaryFile
    BxBinaryFileReader
    BxBinaryFileWriter
  (BxDataObject)
    (BxDataSet)
```

These classes fall into two (or three) groups:

- BinX document-handling classes

Class `BxBinxFile` is a superclass which is responsible for basic behaviour related to the BinX document. In most cases, however, an application programmer will use the extra functionality provided by its concrete subclasses `BxBinxFileReader` and, where writing rather than reading a BinX document, `BxBinxFileWriter` to access a BinX document.

- Binary file-handling classes

Classes `BxBinaryFile`, `BxBinaryFileReader` and `BxBinaryFileWriter` are analogous classes for accessing a binary file.

(Note however, that these classes are not used by the BinX application programmer to explicitly access the binary file. Class `BxBinaryFileReader` is not used at all by the application programmer. Class `BxBinaryFileWriter` is used only to access some of the meta-data about the binary file: file name, block size, etc.).

- Data-handling classes

Subclasses of `BxDataObject` (instances of which are often referred to in the rest of this document as data objects) are responsible for behaviour related to the data structure described in a BinX document. A tree of data objects representing the data structure of the BinX document is built with an object of class `BxDataSet` at its root. This object hierarchy is used as an intermediary layer decoupling the BinX application program from the `BxBinaryFile` object that represents the binary file itself. Thus accessing a particular data item in the binary file is actually done by finding the relevant data object in the `BxDataSet`-rooted hierarchy and using its member functions to access the binary file. This approach makes transparent to the BinX application programmer the raw address calculation, and data type and byte order conversion that would otherwise be necessary to access this data.

When processing a binary file using the BinX library, closely related instances of all three groups of classes described above are required. For straightforward access, however, the application programmer only needs to be concerned with the BinX document-handling group.

To be specific, in order to read a binary file, the application programmer need only explicitly build an object of class `BxBinxFileReader`. The related binary file-handling object (representing the binary file itself), and the data-handling objects (representing the structure of the BinX document) are in turn built by the `BxBinxFileReader` object during its initialisation. The application program obtains access to them through calling `BxBinxFileReader` member functions.

4.2.3 Simple data types

The simple data type classes are as follows:

```
(BXObject)
  (BxDataObject)
    BxInteger
      BxByte8
      BxCharacter8
      BxUnsignedByte8
      BxShort16
      BxUnsignedShort16
      BxInteger32
      BxUnsignedInteger32
      BxLong64
      BxUnsignedLong64
    BxFloat
      BxFloat32
      BxDouble64
      BxString
```

These classes correspond to the simple data types defined in the BinX language as described in section 3.3, ‘The primitive data type elements’ above. These classes are responsible for decoupling the BinX application program from the binary fields of primitive data types in the binary file. Objects of these classes form the leaf nodes on the tree of data objects built to represent the data structure of a BinX document, as described in the previous section.

Note that classes `BxInteger` and `BxFloat` are abstract superclasses which have no methods that are of use to the application programmer and are therefore not documented in the API reference manual. They appear above in order to show the overall class inheritance hierarchy fully.

4.2.4 Complex data types

The complex data type classes are as follows:

```
(BXObject)
  (BxDataObject)
    BxDataset
    BxStruct
    BxUnion
    BxArray
      BxArrayFixed
      BxArrayVariable
```

These classes correspond to the complex data types defined in the BinX language as described in section 3.4, ‘The complex data type elements’ above. These classes are

responsible for decoupling the BinX application program from the binary data of complex data types in the binary file. Objects of these classes form the non-leaf nodes on the tree of data objects built to represent the data structure of a BinX document, as described in the previous section.

The `BxDataset` class is notionally a special subclass of `BxStruct` built to handle the behaviour required of the root of the object tree. In practice, however, no special behaviour is required in the current BinX version, so that `BxStruct` is implemented simply as a C++ typedef for `BxDataset` and the API Reference manual only documents class `BxDataset`. However it is possible that the classes will be implemented differently in future releases of the BinX library, so application programmers are recommended to use the name `BxDataset` only in the definition of the root of the object tree, and to use the name `BxStruct` in the definition of any lower level structures.

4.2.5 Helper classes

```
BxDimension
BxUnionCase
```

These classes are really internal components of the some of complex data type objects described above. They are not visible to the application programmer when reading a BinX document or binary file; however when creating either a BinX document or binary file it is necessary to create these components too.

4.3 An overview of the BinX API

This section is structured in terms of the BinX object containment hierarchy (as opposed to the class inheritance hierarchy, part of which is briefly described in the previous section). The object containment hierarchy is the data structure which the BinX application programmer uses to process the binary file being accessed. It has a direct and natural correspondence with the data structure of both the BinX document describing the binary data file being accessed, and the structure of the binary file itself.

An example of an object containment hierarchy is shown below:

```
BxBinxFileReader
  BxDataset
    BxInteger
    BxInteger
    BxInteger
```

This containment hierarchy describes the data structure used by a BinX application program to process a simple binary file consisting of a single record containing a single sequence of three integers.

The actual containment hierarchy used by a given program depends on the task performed and the data structure of the binary file, however all data containment hierarchies are composed of elements from some or all of the groups of classes described in the previous section, namely:

1. File level elements
2. Primitive data type elements
3. Complex data type elements

These groups also correspond very closely with the 'Document', 'Primitive data type' and 'Complex data type' groups of BinX language elements with which the BinX document

for a given binary file is defined in section 3.1, 'An overview of the BinX language' above.

In the rest of this section we adopt an incremental approach to describing the BinX API. As with the BinX language section, first we show how to process simple binary files, then files with more complex structure. Once this has been covered, we look at more advanced topics: accessing some of the file meta-data from the BinX document in order to be able to do more advanced or generic processing, and error reporting facilities. Finally we show how to create both binary files and BinX documents programmatically.

4.4 Reading a binary file

4.4.1 Opening a BinX document and its associated binary file for input

The following code fragment shows the very small amount of code required to open a BinX document and its associated binary file:

```
#include "BxBinxFileReader.h"
void main() {
    BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
    delete pReader;
}
```

The program above simply creates a new object `pReader`, of class `BxBinxFileReader`. The construction of the object pointed to by `pReader` invokes a substantial amount of complex processing:

- The BinX document file identified (`BinXdoc.xml`) is read and parsed as an XML document using standard XML parsing classes. (Currently BinX uses the Xerces XML SAX parser.)
- The objects produced by the Xerces parser are read by the BinX library, and the binary file identified in the `src` attribute of the `dataset` element of the BinX document is identified and opened.
- The `dataset` element and its contents are used to build a substantial part of the hierarchical structure of data objects (instances of subclasses of `BxDataObject`) described above in section 4.2.2 'File-related classes'.
(Note however that not all the possible data objects are built at this time. For an array, a data object is only built for the first component element of the array. Data objects are built for the other elements in a more 'lazy' fashion, as they are referenced by the application program.)
- The absolute start addresses of these objects are resolved within the binary file. In most cases the BinX library can calculate these addresses based on information contained on the BinX document about sizes and dimensions. In the case of a variable array, however, this cannot be done solely by looking at the BinX document and the binary file itself is accessed to determine the cardinality of the outermost dimension of the array.

Once this initialisation has been done, the BinX application programmer can access the root of the hierarchical structure of data objects by calling the `getDataset()` member function of the `BxBinxFileReader` instance. This is demonstrated in the following code fragment which extends that above:

```
#include "BxBinxFileReader.h"
void main() {
```

```

BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
BxDataset* pDataset = pReader->getDataset();
//
// now we can read the binary file
// using the data object hierarchy
//
delete pReader;
}

```

4.4.2 Reading sequentially from a simple binary file

Once a BinX document and its associated binary file have been opened, the data in the binary file can be simply accessed using the data object hierarchy already built during `BxBinxFileReader` object initialisation. For primitive data objects access is achieved by invoking a member function of each data object whose value is required.

The following example builds on that in the preceding section to show how to access sequentially the data in a simple binary file consisting of a single record. The binary file is described by the example XML document in section 3.3, 'The primitive data type elements' above.

```

#include "BxBinxFileReader.h"
void main() {
    BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
    BxDataset* pDataset = pReader->getDataset();
    //
    // now read the binary file
    // using the data objects in the dataset structure
    //
    if(pDataset->next()) {
        BxDataObject* pCurrencySymbol = pDataset->getDataObject();
        bx_char8 currencySymbol = pCurrencySymbol->getChar();
    };
    //
    //
    if(pDataset->next()) {
        BxDataObject* pVatRate = pDataset->getDataObject();
        bx_short16 vatRate = pVatRate->getShort();
    };
    //
    //
    if(pDataset->next()) {
        BxDataObject* pStdIncTaxRate = pDataset->getDataObject();
        bx_short16 stdIncTaxRate = pStdIncTaxRate->getShort();
    };
    //
    //
    if(pDataset->next()) {
        BxDataObject* pGDP = pDataset->getDataObject();
        bx_int32 gDP = pGDP->getInt();
    };
    //
    //
    if(pDataset->next()) {
        BxDataObject* pLandArea = pDataset->getDataObject();
        bx_float32 landAreaInAcres = pLandArea->getFloat();
    };
    //
    //
    if(pDataset->next()) {

```

```

    BxDataObject* pAnnualRainfall = pDataset->getDataObject();
    bx_float32 annualRainfallInInches =
        pAnnualRainfall->getFloat();
    };
    //
    //
    if(pDataset->next()) {
        BxDataObject* pAvgeSunshine = pDataset->getDataObject();
        bx_float32 averageDailySunshineInHours =
            pAvgSunshine->getFloat();
    };
    delete pReader;
}

```

The program reads through the sequence of primitive data objects contained in the `BxDataset` object (which is itself of complex type). This is done using two sets of behaviour:

- The object of class `BxDataset` keeps a current component object as part of its internal state. The `BxDataset::next()` member function advances the `BxDataset` object's current component pointer to point to the next component object in the data structure. The `BxDataset::getDataObject()` member function is then used to return a pointer to the actual data object representing the current component. More detail about how the BinX library can be used to navigate and access objects of complex data types is given below in section 4.4.4, 'Accessing complex data types'.
- For each primitive data object thus obtained, a data-type-specific member function `getDataType()` is invoked to extract the binary data described by the data object. (The program text `dataType` should be substituted, as described in section 1.4, 'Notations used in this manual', by an actual data type. Details of actual data types available are given in Appendix B.1, 'Primitive data type protocol'. A general description of how the BinX library can be used to access primitive data types is given in the next section).

4.4.3 Reading from a simple binary file in random order

When reading individual elements from a binary file in random order, it is necessary both to locate the relevant element in the data object hierarchy, and also to reposition the binary file pointer, to point to the binary field corresponding to the element in the data object hierarchy, before obtaining the data from the file.

A modified version of the example in the previous section is shown below. It gives examples of direct access to component objects in the same file as that scanned sequentially in the previous example:

```

#include "BxBinxFileReader.h"
void main()
{
    BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
    BxDataset* pDataset = pReader->getDataset();
    //
    // now read the binary file
    // using the data objects in the dataset structure
    //
    //
    // first access an element by index
    //

```

```

int gDPIndex = 3;
BxDataObject* pGDP = pDataset->getDataObject (gDPIndex) ;
//
// move the binary file pointer to the correct position
pGDP->locate () ;
//
bx_int32 gDP = pGDP->getInt () ;
//
delete pReader;
}

```

The program works using three sets of behaviour:

- The `BxDataset::getDataObject ()` member function is used to return a pointer `pGDP` to the actual data object representing the desired component (in this case the fourth element of the structure).
More detail about how the BinX library can be used to navigate and access objects of complex data types is given below in section 4.4.4, 'Accessing complex data types'.
- The binary file is then re-positioned to the object referenced by the pointer using the `BxDataObject::locate ()` method.
- A data-type-specific member function, in this case `getInt ()` is invoked to extract the binary data described by the data object. A general description of how the BinX library can be used to access primitive data types is given in the next section).

4.4.4 Accessing primitive data types

The BinX library supports a large number of primitive data types in the binary data file. (These are defined in the BinX language; see section 3.3, 'The primitive data types' above for more details.) In order to write a BinX application program, it is necessary for the programmer to have available a set of application programming primitives to allow data of each type to be extracted from the binary file, converted into a suitable C++ data type, etc.

For each BinX language primitive data type, the BinX library provides the application programmer with the following programming constructs:

- An implicit C++ native data type into/from which the BinX primitive data type can be converted.
(For example the `<character-8>` data type is converted to/from the C++ `char` native data type.)
- A `typedef` for the implicit C++ data type. In BinX application programs this `typedef` should be used in place of the native C/C++ data types.
(For example the type `bx_char8` is defined as (a synonym for) the C++ native data type `char` to be used for converting to/from `<character-8>` data.)
- A C++ data object class (i.e. a subclass of `BxDataObject`) which decouples the application program from the binary data in the file as described above in this section.
(For example class `BxCharacter8` is provided to wrap data of type `bx_char8`)
- A constructor function which constructs an object of the appropriate C++ data object class to wrap an instance of one of the C++ native data types.
(For example class `BxCharacter8` has constructor `BxCharacter8::BxCharacter8 (bx_char8)`.)

- Getting and setting member functions (`getDataType()` and `setDataType()`) for a `BxDataObject` instance representing a primitive data object through which the data in the binary data file can be accessed
(For example `BxDataObject::getChar()` and `BxDataObject::setChar()` are provided for getting and setting `<character-8>` values.)
Note that calling the `BxDataObject::getDataType()` member function will result in reading from the binary data file. If the data object has no binary data file associated with it, then zero will be returned.

The BinX library internally represents the type of each data object with an integer constant value. The following additional protocol is available to enable the application programmer to take advantage of this:

- A symbol which identifies the constant value used by the BinX library to represent the data type.
(For example, the value used for `<character-8>` data is `BXCHARACTER8`.)
- The `BxDataObject::dataClass()` member function returns the BinX library internal type value for the object.
(For example, the value returned by `BxDataObject::dataClass()` on a data object of class `BxCharacter8` is `BXCHARACTER8`.)
- A comparison macro which can be used to determine the type of data represented by any `BxDataObject` instance.
(For example the comparison macro for `<character-8>` data is `ISCHARACTER8()`.)
Thus the following code fragment can be used to determine the data type of an object:

```
BxCharacter8* pChar = new BxCharacter8("c");
BxDataObject* pdo = pChar;
if (ISCHARACTER8(pdo->dataClass())) (
    Printf("Data object pdo is a character.");
```

With the exception of the `dataClass()` member function (which is not data-type-specific), Appendix B.1, 'Primitive data type protocol' below lists the type-specific instances of the above protocol which are provided for all primitive data types supported.

4.4.5 Accessing String data type

For the string data types, which are the fixed length string, the variable length string and the delimited string (a string terminated with a specified ASCII character), the BinX library provides the application programmer with the following programming constructs:

- A C++ data object class (i.e. `BxString`, which is a subclass of `BxDataObject`) which decouples the application program from the binary data in the file. The internal representation of the data in `BxString` is an array of characters.
- Three constructor functions which construct a `BxString` object to wrap the array of characters. The constructor `BxString::BxString(char*, int)` constructs a fixed length string. The constructor `BxString::BxString(char*, BxInteger*)` constructs a variable length string and the constructor `BxString::BxString(char*)` constructs a delimited string with the default terminating character. The delimiter string can be modified by the `BxString::setDelimiter(char*)` function.

- The data in the binary data file can be accessed via the `BxString::toString(bool)` member function. The function returns the XML element or just the data value depending on the value of the Boolean. The data in the binary data file can be modified with the `BxString::setText(char*)` member function.
- A symbol which identifies the constant value used by the BinX library to represent the three string data types. The symbol for a fixed length string is `BXSTRINGFIXED`, the symbol for a variable length string is `BXSTRINGVARIABLE` and the symbol for a delimited string is `BXSTRINGDELIMITED`. There is also the symbol `BXSTRING`, which identifies a string in general.
- The `BxString::getType()` member function returns the BinX library internal type value for the object.
- A comparison macro which can be used to determine the type of data represented by any `BxDataObject` instance. The comparison macro for a fixed length string is `ISSTRINGFIXED()`, the comparison macro for a variable length string is `ISSTRINGVARIABLE()` and the comparison macro for a delimited string is `ISSTRINGDELIMITED()`. There is also the comparison macro `ISSTRING()`, which identifies a string in general.

4.4.6 Accessing complex data types

For the complex data types, the BinX library adopts a different approach to accessing data. Unlike the primitive types, for the complex types the BinX library does not support a corresponding C++ native data type into/from which BinX complex types can be converted. For complex data types, therefore, the main purpose of the BinX library API is navigating the complex data object's structure to reach its component data object(s).

For most types of data structure, the navigational facility provided is the ability to iterate through the elements of the data structure sequentially, in the order in which components are stored in the binary file. The subsections immediately following give more details about the facilities available for each type of data structure.

Once reached, each component data object may itself be primitive or complex:

- For component objects of primitive data type, the primitive data type API described in the preceding section can be used to access the object.
- For component objects of complex data type, the complex data type API can be used recursively to navigate the component object itself.

As well as the navigational behaviour, the BinX library also provides some of the same protocol for complex data types as it does for primitive data types, notably type identifiers and comparison macros. These are documented in Appendix B.2, 'Complex data type protocol' below.

Note also that as a general rule, the relevant navigational member functions of the complex data types return values which are component data object references rather than copies. These objects should not, therefore be deleted by the application programmer. There is, however, an exception to this rule - objects of class `BxArray` and its subclasses return copies of component objects which it is the application programmer's responsibility to delete.

4.4.6.1 Accessing structures (including datasets)

Sequential access to component data objects

Both structures and datasets are represented by the `BxDataset` class. The simplest approach to navigating through a structure is to use the `BxDataset::next()` member function. The first time this is called for a data object representing a given structure, it sets a 'current component object' pointer within the data object to point to the first data object in the structure. Subsequent calls increment the pointer to point to succeeding data objects. The pointer can be reset to the start of the structure with the `BxDataset::reset()` member function. Once a component object has been located, the `BxDataset::getDataObject()` member function will return a pointer to the current component. This takes the form of a pointer to an object of class `BxDataObject`. An example of a program using this behaviour has already been given in section 4.4.3, 'Reading from a simple binary file'.

Direct access to component data objects

Component objects within a structure can also be accessed directly, rather than sequentially, using one of the following approaches:

- The `BxDataset::getDataObject(n)` member function locates the $n+1$ 'th component data object of the structure (starting from element 0 not 1).
- The `BxDataset::getDataObject(eltName)` member function locates the first data object in the structure with name `eltName`. (Recall that a data object's name is declared using the `varname` attribute of the element corresponding to the object in the BinX document associated with the file.)

A modified version of the example in section 4.4.2, 'Reading from a simple binary file' is shown below. It gives examples of direct access to component objects in the same file as that scanned sequentially in the previous example:

```
#include "BxBinxFileReader.h"
void main()
{
    BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
    BxDataset* pDataset = pReader->getDataset();
    //
    // now read the binary file
    // using the data objects in the dataset structure
    //
    //
    // first access an element by index
    //
    int gDPIndex = 3;
    BxDataObject* pGDP = pDataset->getDataObject(gDPIndex);
    //
    // move the binary file pointer to the correct position
    pGDP->locate();
    //
    //
    bx_int32 gDP = pGDP->getInt();
    //
    // now access an element by name
    //
    char[3] gDPName = "GDP";
    BxDataObject* pVatRate = pDataset->getDataObject(gDPName);
```

```

//
// move the binary file pointer to the correct position
pVatRate->locate();
//
//
bx_short16 vatRate = pVatRate->getShort();
//
delete pReader;
}

```

Type-specific access to component objects

All of the member functions in the previous parts of this subsection have return type `BxDataObject*`. In order to perform type-specific operations on the object to which it refers, the pointer must be explicitly cast into a pointer to the actual specific subclass.

For primitive data types, this is rarely necessary - almost all operations on primitive data types are actually implemented as member functions of the superclass `BxDataObject`, and those that are implemented differently for specific subclasses tend to be implemented using virtual member functions of `BxDataObject`.

For objects of complex data types, however, the situation is different. These implement major extensions to the abstract superclass protocol, and thus it is normally necessary to obtain a pointer to an object of the actual specific subclass to use the extended behaviour. Although this process can be performed with a (type) cast, another set of type-specific member functions is available to make this process slightly simpler. The set of member functions:

- `BxDataset::getDataObject()`
- `BxDataset::getDataObject(n)`
- `BxDataset::getDataObject(eltName)`

perform analogous functions to the `getDataObject()` member functions described above, but return pointers to objects of specific `BxDataObject` subclasses, rather than pointers to objects of `BxDataObject` itself.

The following table gives the relevant `BxDataset` member function names for the complex data types:

Data type	<code>BxDataset</code> member functions
<code>BxDataset</code>	<code>BxDataset* getDataset()</code> <code>BxDataset* getDataset(int index)</code> <code>BxDataset* getDataset(const char* eltName)</code>
<code>BxArray</code>	<code>BxArray* getArray()</code> <code>BxArray* getArray(int index)</code> <code>BxArray* getArray(const char* eltName)</code>
<code>BxUnion</code>	<code>BxUnion* getUnion()</code> <code>BxUnion* getUnion(int index)</code> <code>BxUnion* getUnion(const char* eltName)</code>
<code>BxString</code>	<code>BxString* getString()</code> <code>BxString* getString(int index)</code> <code>BxString* getString(const char* eltName)</code>

Note that there is no member function that returns an object of type `BxArrayFixed*` or `BxArrayVariable*`. This is rarely required, for, just as in the more general case above, the various specific subtypes mostly just implement the virtual functions defined in the

abstract superclass `BxArray`. However class `BxArrayVariable` also extends the `BxArray` protocol with member functions `getSizeObject` and `setSizeObject`. In this case an object of type `BxArray*` must be explicitly cast to type `BxArrayVariable*` to invoke these functions as shown in the following code fragment:

```
BxArrayFixed* pArrayFixed = (BxArrayFixed*) pDataset->getArray();
```

4.4.6.2 Accessing unions

Unions are represented by the `BxUnion` class. Unlike the other complex data types, a union represents a set of alternative components rather than a group of coexisting components. Hence navigating a union structure is mostly a question of accessing the single element which is present in the union object. This is simply achieved using the member function `BxUnion::getDataObject()`. In addition it may also be relevant to look at the value of the discriminant of the union; this determines the element of the union which is present. This can be done using the `BxUnion::getDiscriminant()` member function. This returns the data object for the discriminant of the union. This can be interrogated using the primitive object protocols described above in section 4.6, 'Accessing Primitive Data Types', to determine the value that is present.

4.4.6.3 Accessing arrays

Arrays are represented by the abstract superclass `BxArray` class and its concrete subclasses `BxArrayFixed`, and `BxArrayVariable`. As mentioned earlier, the behaviour of arrays is mostly defined as a set of virtual functions in `BxArray`, and where not generic is implemented in the concrete subclasses.

For any type of array, the BinX library supports two different views of the array:

- A (one-dimensional) ordered collection of component elements, sequenced according to the order in which they occur in the binary file. This can be accessed either sequentially or directly.
- A multi-dimensional array of component elements, structured according to the dimensions specified in the BinX language document that defines the file.

Sequential access to component data objects

Component objects in an array are accessed sequentially using a somewhat similar protocol to that used for accessing elements of a structure sequentially. The `BxArray::getNext()` member function is used. The first time that this is called it returns the first data object in the array, succeeding calls return succeeding data objects in the array viewed as a flat single dimensional structure. The `BxArray::toFirst()` member function can be used to reset the current position in the array to the start of the array if required. Readers should note the differences between this protocol and that for structures. These include the following:

- `BxArray::getNext()` returns an object of type `BxDataObject*`, whereas `BxDataset::next()` returns a Boolean indicating success or failure, which must be followed by `BxDataset::getDataObject()` to return the object of type `BxDataObject*`.
 - `BxArray::toFirst()` is used to reset the current position to the start of the array, whereas `BxDataset::reset()` is used to reset the current position to the start of a structure.
 - The data object pointed to by `BxArray::getNext()` is constructed especially to be returned to the application, and should be deleted by the application program when
-

no longer required, whereas the data object pointed to by `BxDataset::getDataObject()` is managed by the BinX library, and should not be deleted by the application program.

Direct access to component data objects

As mentioned above, direct access to components is possible viewing the array either as a single dimensional array, regardless of any multi-dimensional structure declared in the BinX document describing the file, or using the multi-dimensional structure explicitly.

To access any array as a single-dimensional array the `BxArray::get(dimIndexInt)` member function is used.

To access an array using the multi-dimensional structure (if declared) the following member functions are used, depending on the dimensionality declared for the array:

- For 2-, 3- and 4-dimensional arrays, the following member functions are used:
 - `BxArray::get(dim2IndexInt, dim1IndexInt)`
 - `BxArray::get(dim3IndexInt, dim2IndexInt, dim1IndexInt)`
 - `BxArray::get(dim4IndexInt, dim3IndexInt, dim2IndexInt, dim1IndexInt)`
- For arrays of higher dimensionality, member function `BxArray::getMore()` is used. This has a variable length argument list, so that any number of index dimensions can be specified.

Type-specific access to component objects

All of the above member functions return objects of class `BxDataObject` to the caller. The same issues arise with regard to calling the type-specific behaviour of specific subtypes of component data object as are documented in section 4.7.1, 'Accessing structures (including datasets)' with components of structures. In the case of arrays, however, there are no equivalents of the `BxDataset::getDataType()` member functions provided. The caller must simply cast the object into the correct subclass of `BxDataObject` in order to invoke specialised behaviour.

4.4.7 Access to binary file meta-data

A BinX application program processing a binary file has access to some of the meta-data items that the BinX library uses to correctly process the binary data fields. These are typically closely related to, or derived from, the meta-data stored in the BinX document describing the binary file. This information is particularly useful for writing more generalised, or more complex programs which deal with more than one binary data file. However such information should be used with care – it is possible to use it to bypass the BinX library files to access raw data but this sort of use is neither recommended nor supported.

In general for each data object the following information is available:

(Note: The following applies to instances of subclasses of `BxDataObject`. It does not apply to instances of either the file handling classes `BxBinxFile` and `BxBinaryFile` or the helper classes `BxDimension` and `BxUnionCase` or to any of their subclasses):

- The type of the data object.
This is given by the `BxDataObject::dataClass()` member function. The member function itself returns an integer identifying the type. A set of comparison macros are supplied (documented in Appendices B.2 and B.3) which can be used to determine if

the object is of a particular type. A sample program demonstrating this is shown below. The program is a part of the example in section 4.5, 'Reading from a simple binary file' enhanced with data type-testing logic so that the type of the object is checked before a type-specific member function is invoked:

```
#include "BxBinxFileReader.h"
void main() {
    BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
    BxDataset* pDataset = pBinx->getDataset();
    //
    // now read the binary file
    // using the data objects in the dataset structure
    //
    if(pDataset->next()) {
        BxDataObject* pCurrencySymbol = pDataset->getDataObject();
        if (ISCHARACTER8(pCurrencySymbol->dataclass())) {
            bx_char8 currencySymbol = pCurrencySymbol->getChar();
        };
        delete pReader;
    }
}
```

- The `varName` attribute of the BinX document element describing the data object.
This is given by the `BxDataObject::varName()` member function.
- The size of the binary data field described by the object.
- The `BxDataObject::size()` member function returns the size of the field in bytes including any padding to ensure that particular component objects are properly aligned for computation over a data value.
- The byte order of the object.
This is given by the `BxDataObject::byteOrder()` member function.

Note that objects of class `BxDataset` behave differently with respect to meta-data depending on whether or not the `BxDataset::next()` member function has been called (and thus a current object pointer set internally). If the current object pointer has not been set, then the object of class `BxDataset` will return meta-data describing the complete dataset object, but if the current object pointer has been set then the equivalent meta-data about its current component object is returned instead. The same is not true for objects of class `BxArray` or any subclass of `BxArray`.

In addition, for specific subtypes of data object there are specific items of meta-data available. These are described in the following subsections.

4.4.7.1 Specific `BxDataset` meta-data

- The number of component objects contained in the object.
This is given by the `BxDataset::count()` member function.

4.4.7.2 Specific `BxUnion` meta-data

- The value of the discriminant of the union.
This is given by the `BxUnion::getDiscriminant()` member function.

4.4.7.3 Specific `BxArray` meta-data

- The number of dimensions to an array index.
This is given by the `BxArray::getDimensions()` member function.

- The end value for the index range for each dimension of the array. This is given by the `BxArray::getIndexTo(dimensionOrdinal)` member function. This is true both for fixed and variable size arrays. In the case of the variable size array, the variable size dimension will always be the last dimension.

4.5 Runtime errors and warnings

The BinX library provides internally a lightweight protocol to standardise the way that runtime errors and warnings are reported. When part of the library discovers a runtime error, or wishes to issue a warning message, a message is written to standard output. For errors, the text of the message is standardised. For warnings, the text is specified by the invoking routine in the library.

In addition, for errors the error code of the error encountered is stored (temporarily) in the object encountering the error. This can be accessed using the `BxObject::getErrorCode()` member function. Once obtained, the text associated with the error code can be obtained using the static member function `BxObject::getErrorMessage()`. Application programmers should note that invoking the `BxObject::getErrorCode()` member function of an object has a side effect of resetting the error code stored in the object to 0.

A sample program demonstrating the use of these functions is shown below, the program is a part of the example in section 4.5, 'Reading from a simple binary file' enhanced with error handling logic:

```
#include "BxBinxFileReader.h"
void main() {
    BxBinxFileReader* pReader = new BxBinxFileReader("BinXdoc.xml");
    BxDataset* pDataset = pBinx->getDataset();
    //
    // now read the binary file
    // using the data objects in the dataset structure
    //
    if(pDataset->next()) {
        BxDataObject* pCurrencySymbol = pDataset->getDataObject();
        bx_char8 currencySymbol = pCurrencySymbol->getChar();
        if (currencySymbol == 0) { // possible error
            if (pCurrencySymbol->getErrorCode() != 0){ // actual error
                // analyse error and take appropriate action
            }
        }
    }
    delete pReader;
}
```

The library does not have a standard way to communicate error conditions to the caller, although generally speaking this is done using null or zero return values.

The BinX API Reference manual contains details of any values that are returned to a BinX application program which denote error conditions. If a BinX application program receives such return values, these member functions can be used to obtain additional information regarding the error code and error message.

4.6 Creating a binary file

The BinX library's approach to writing data, either to a binary file or to its associated BinX document, is as follows:

- Firstly all the data necessary to write both the binary file, and to describe its structure in the associated BinX document, must be assembled in application memory and linked together into a single data structure.
This structure is more or less the same as the one which the library would progressively build if it were later used to read the binary file and its associated BinX document.
- Once this has been done, data can then be written to either or both of the binary file and its associated BinX document.
(There is no dependency between the file and document writing processes, but both are dependent on the existence of the structure of data objects from which to generate the file or document data to be written.)

This approach obviously places limitations on the size of binary file which can be written.

This also means that the BinX library API to handle the process of creating a new binary file is quite different from the API to read an existing binary file:

- The main work is to create the individual data objects, containing both the data for the binary file and the meta-data for the BinX document, and link them together.
This is a reasonably complex task, and itself breaks down into two parts:
 - Build data objects of primitive data types
 - Build data objects of complex data types from component data objects of both primitive and complex data types.
This progressively assembles the tree of data objects, starting with the leaves and working upwards to the root. (This is the tree of data objects previously described above in section 4.2.2, 'File-related classes' under 'Data-handling classes'.)
- Once the linked structure of data objects has been created in memory, the data can be written to the binary file, without further object building or linking.
This is a simple and straightforward task.
- Once the linked structure of data objects has been created in memory, the BinX document that describes the file can also be written.
To do this objects representing the binary file and the BinX document (as described above in section 4.2.2, 'File-related classes' under 'Binary file-handling classes' and 'BinX document-handling classes') must be built, and then appropriate member functions invoked.

More details about these processes are contained in the following sections

4.6.1 Creating a structure of data objects in memory

As mentioned above, what needs to be built is the tree of data objects (each of which is an instance of a subclass of `BxDataObject`) which has already been described at some length in section 4.2.2, 'File-related classes' under 'Data-handling classes'.

Essentially this is done by progressively constructing the data objects individually and then linking them together into the complete tree by assembling them into progressively

larger complex objects. (However in order to build some of the complex data objects correctly it is sometimes also necessary to construct subcomponent objects of helper classes with which to initialise them. These were mentioned in section 4.2, 'BinX class inheritance structure', but have not been dealt with before because they are not visible to the application programmer when using the 'reader' part of the API.)

The following sections describe building the tree from the leaves upwards to the root. We start with building the primitive data objects, and then move on to building the non-leaf nodes on the tree (the complex data objects) from individual component data objects.

4.6.1.1 Creating objects of primitive data types

Building objects of primitive data types (the leaves on the data object tree) is simple and straightforward. It is done by constructing a new object of the appropriate subclass of `BxDataObject`, using as parameter an instance of the appropriate C++ programming language native data type.

(Recall from section 4.43, ("Accessing primitive data types"), that the primitive data type protocol implemented by the BinX library includes:

- An implicit C++ native data type, together with a C++ typedef for it to use in application programming.
- A constructor to build a data object of the appropriate class from an object the implicit C++ native data type

Appendix B.1, 'Primitive data type protocol' gives details of the classes, typedefs, and constructor functions for each of the BinX primitive data types.)

Once each data object has been constructed, its byte order and variable name can be set using the member functions `BxDataObject::setByteOrder()` and `BxDataObject::setVarName()` respectively. Note that member function `BxDataObject::setByteOrder()` takes a parameter of type `BxByteOrder`. This is an enumeration defined in the `BxBinaryFile` class header file.

4.6.1.2 Creating objects of complex data types from component objects

In order to create a complex data object the following processing tasks must be performed:

- Create an empty instance of the complex data object itself (using a zero parameter constructor function)
- Where necessary, initialise the empty complex data object with appropriate structural information.
- Add the component data objects.
- Set the byte order and variable name, if required.

Some of these tasks are subtly different for each complex object type, so the next few sections deal separately with the process for structures, arrays and unions respectively.

4.6.1.3 Creating structure objects (including datasets)

Structures are the simplest of the complex data objects to create. Creating a structure involves the following:

- Create an empty structure instance using the zero parameter constructor
`BxDataset::BxDataset()`.
- Add the component data objects using member function
`BxDataset::addDataObject()`.
- Set the byte order and variable name for the structure, if required, using member functions `BxDataObject::setByteOrder()` and `BxDataObject::setVarName()` respectively.

An example code fragment to do create a structure of primitive data objects is given below:

```
//
// Create primitive data objects
//
BxShort16* pDataObj1 = new BxShort16();
BxInteger32* pDataObj2 = new BxInteger32(100);
BxLong64* pDataObj3 = new BxLong64();
//
// Create an empty structure
//
BxDataset* pStructObject_ = new BxDataset();
//
// Add primitive data objects to the structure
//
pStructObject->addDataObject(pDataObj1);
pStructObject->addDataObject(pDataObj2);
pStructObject->addDataObject(pDataObj3);
```

4.6.1.4 Creating array objects

Creating an array involves the following

- Create an empty array instance using the zero parameter constructor. For a fixed array this is `BxArrayFixed::BxArrayFixed()`.
- Initialise the structural meta-data for the array:
 - The element type
This is set with the `BxArray::setElementType()` member function.
(The parameter value is a sample data object of the type to be added to the array)
 - The dimensions are set using the following procedure:
 - Set the number of dimensions using `BxArray::setDimensions()`
 - For a variable size array, set the type and value of the size reference field using `BxArrayVariable::setSizeObject()`.
 - For each dimension:
 - Construct a `BxDimension` object using constructor `BxDimension::BxDimension(0, toIndex, dimName)`
 - Add the dimension to the array using `BxArray::setDimension(dimNo, dimension)`
- Add the elements to the array using `BxArray::addElement()`

- Set the byte order and variable name for the array, if required, using member functions `BxDataObject::setByteOrder()` and `BxDataObject::setVarName()` respectively.

An example code fragment to create an array of primitive data objects is given below:

```
//
// Create the primitive data objects
//
BxInteger32*[5][2] pPrimitiveDataArray = {
    {new BxInteger32(1), new BxInteger32(2)},
    {new BxInteger32(3), new BxInteger32(4)},
    {new BxInteger32(5), new BxInteger32(6)},
    {new BxInteger32(7), new BxInteger32(8)},
    {new BxInteger32(9), new BxInteger32(10)}
}
//
// Create an empty ArrayFixed instance
//
BxArrayFixed* pArrayObject = new BxArrayFixed();
//
// Set the element type of the BxArray
//
BxInteger32* elemObj = new BxInteger32(0);
pArrayObject->setElementType(elemObj);
//
// Set the no of dimensions and dimensions
//
pArrayObject->setDimensions(2);
BxDimension* pDim0 = new BxDimension(0,1,"Inner");
pArrayObject->setDimension(0, pDim0);
BxDimension* pDim1 = new BxDimension(0,4,"Outer");
pArrayObject->setDimension(1, pDim1);
//
// Add the components of the array
//
for (int i=0; i<5; ++i) {
    for (int j=0; j<2; ++j)
        pArrayObject->addElement(pPrimitiveData[i][j]);
}
```

4.6.1.5 Creating union objects

Creating a union involves the following:

- Create an empty union instance using the zero parameter constructor `BxUnion::BxUnion()`.
- Add the individual data objects, together with the discriminant values that identify them, to the union as individual cases using `BxUnion::addUnionCase()`. (Recall that all discriminant values must be data objects of the same type, which must be one of `BxByte8`, `BxCharacter8`, `BxShort16`, `BxInteger32`, `BxLong64` or the four equivalent unsigned integer types.)
- Set one of the cases as the assigned case, i.e. the case that identifies the object present in this instance of the union. This is achieved by setting the actual discriminant value of the array using member functions `BxUnion::setDiscriminant()` followed by `BxUnion::setAssignedCase()`.

- Set the byte order and variable name for the union, if required, using member functions `BxDataObject::setByteOrder()` and `BxDataObject::setVarName()` respectively.

An example code fragment to create a union of primitive data objects is given below:

```
//
// Create the primitive data objects
//
BxShort16* pDataObj1 = new BxShort16();
BxInteger32* pDataObj2 = new BxInteger32(100);
BxLong64* pDataObj3 = new BxLong64();
//
// Create the discriminant values for the primitive data objects
//
BxByte8 * pDiscrimVal1 = new BxByte8(1);
BxByte8 * pDiscrimVal2 = new BxByte8(2);
BxByte8 * pDiscrimVal3 = new BxByte8(3);
//
// Create an empty union
//
BxUnion * pUnionObject_ = new BxUnion();
//
// Add cases to the union object
//
pUnionObject_>addUnionCase(pDiscrimVal1, pDataObj1);
pUnionObject_>addUnionCase(pDiscrimVal2, pDataObj2);
pUnionObject_>addUnionCase(pDiscrimVal3, pDataObj3);
//
// Create the actual discriminant value
//
BxByte8 * pActualDiscrim = new BxByte8(pDiscrimVal2);
//
// Set the union's assigned case from the actual discriminant value
//
pUnionObject->setDiscriminant(pActualDiscrim);
pUnionObject_>setAssignedCase();
```

4.6.2 Creating a new binary file with an existing BinX document

Once a tree structure of data objects has been created (as described in the previous section) with an instance of class `BxDataset` as its root, the member function `BxDataset::toStreamBinary(FILE*)` can be used to write the structure to a binary file.

4.6.3 Creating a new associated BinX document

Once a tree structure of data objects has been created (as described in section 4.6.1, 'Creating a structure of data objects in memory') with an instance of class `BxDataset` as its root, the following processing needs to be performed to create the BinX document that describes the data structure:

- Create linked instances of `BxBinxFileWriter` and `BxBinaryFileWriter` and link them to the existing tree structure of data objects. This involves the following:
 - Create an instance of class `BxBinxFileWriter` pointing to the file which will contain the BinX document when written using `BxBinxFileWriter::BxBinxFileWriter(docFileName)`.

- Create an instance of class `BxBinaryFileWriter` with a dummy file name using:
`BxBinaryFileWriter: BxBinaryFileWriter()` and then
`BxBinaryFileWriter::copyFileName(binFileName)`.
 (The `BxBinaryFileWriter` instance will not be used as a destination for data, but is required as an intermediate link between the `BxBinxFileWriter` instance and the `BxDataset` instance at the root of the tree of data objects.)
- Set the `BxBinxFileWriter` instance to point to the `BxBinaryFileWriter` instance using
`BxBinxFileWriter::setBinaryFilePtr(pBinFileWriter)`.
- Set the `BxBinaryFileWriter` object to point to the `BxDataset`-rooted structure already built using
`BxBinaryFile::setDatasetPointer(pDataset)`.
- Call the `BxBinxFileWriter::save()` member function to create the BinX document with the designated document file name.

The following code example illustrates this:

```
// (Assume pDataset has already been created as a pointer to
// a BxDataset object at the root of a structure of objects)
//
// Create a BxBinxFileWriter instance
//
BxBinxFileWriter* pWriter = new BxBinxFileWriter("output.xml");
//
// Create a binary file
//
BxBinaryFileWriter* pDataFile = new BxBinaryFileWriter();
pDataFile->copyFileName("dummy.bin");
//
// Attach the binary file to the BinxFileWriter instance
//
pWriter->setBinaryFilePtr(pDataFile);
//
// Set the binary file to point to the BxDataset-rooted structure
//
pDataFile->setDatasetPointer(pDataset); // assume pDataset
//
// Now the BinX document is associated with the dataset structure,
// Save the document
//
pWriter->save();
//
delete pWriter;
```

Appendices



Appendix A BinX Language Reference

The BinX language is documented here with a list of language elements (tags) in alphabetic order. For each element / tag the following are given:

- A list of attributes and their valid values
- A sequence of component elements and their value types

When an attribute or component element has additional properties, i.e. it is mandatory, or of restricted values, these are also given. By default all attributes are optional, and all component elements are optional and of unlimited cardinality.

A.1 Language Elements

- `<arrayFixed>`

Attributes

<code>byteOrder</code>	Either "bigEndian" or "littleEndian"
<code>varName</code>	A text string specifying a name that identifies this element in the BinX API.

Component elements

<code><anyDataType></code>	Mandatory. Exactly one required. Describes the element that is collected in the array, i.e. any of: <code><arrayFixed></code> <code><arrayVariable></code> <code><byte-8></code> <code><character-8></code> <code><double-64></code> <code><float-32></code> <code><integer-32></code> <code><long-64></code> <code><short-16></code> <code><struct></code> <code><union></code> <code><unsignedByte-8></code> <code><unsignedInteger-32></code> <code><unsignedLong-64></code> <code><unsignedShort-16></code>
<code><dim></code>	Mandatory. Exactly one required.

- `<arrayVariable>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

Component elements

<anyDataType>	Mandatory. Exactly one required. Describes the element that is collected in the array.
<dim>	Mandatory. Exactly one required. The outermost <dim> element may not have indexTo attribute specified.
<sizeRef>	Mandatory. Exactly one required.

- <binx>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
version	must be "1.0"
xmlns	Mandatory. Should always be "http://www.edikt.org/binx/2003/06/binx" A text string specifying an identifier for the BinX schema used.

Component elements

<definitions>	Optional. If present, exactly one required.
<dataset>	Mandatory. Exactly one required.

- <byte-8>

Attributes

varName	A text string specifying a name that identifies this element in the BinX API.
---------	---

- <case>

Attributes

discriminant Value	A text string specifying the value of the discriminant that identifies an occurrence of this case of the union.
--------------------	---

Component elements

<anyDataType>	Mandatory. Exactly one is required. The data type element of the union that occurs in this case.
---------------	---

- <character-8>

Attributes

varName	A text string specifying a name that identifies this element in the BinX API.
---------	---

- <dataset>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
-----------	--------------------------------------

varName	A text string specifying a name that identifies this element in the BinX API.
src	Mandatory. A text string specifying the path name of the binary file whose structure is encoded in this document.

Component elements

<anyDataType>	A sequence of any complex or primitive data elements
---------------	--

- <definitions>

Component elements

<defineType>	A sequence of 1 or more <defineType> elements.
--------------	--

- <defineType>

Attributes

typeName	Mandatory. A text string specifying the name of the user-defined type.
----------	---

Component elements

<actualType>	Mandatory. Exactly one required. An (anonymous) complex data type element specifying the actual type of the user-defined type name
--------------	---

- <dim>

Attributes

name	A text string specifying the name of the index dimension.
indexTo	A text string specifying the highest index value for the (zero-based) index dimension.

Component elements

<innerDim>	No more than one allowed. A further <dim> element specifying any inner dimensionality to the array.
------------	--

- <discriminant>

Component elements

<discrimDataType>	Mandatory. Exactly one required. An element whose value identifies the case of any instance of the union. It should be of one of the following types: <byte-8> <character-8> <integer-32> <long-64> <short-16>
-------------------	--

	<unsignedInteger-32> <unsignedLong-64> <unsignedShort-16>
--	---

- <double-64>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- <float-32>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- <integer-32>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- <long-64>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- <short-16>

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- <sizeRef>

Component elements

<integer DataType>	Mandatory. Exactly one required.
-----------------------	----------------------------------

- <string>

Attributes

delim	A text string specifying the ASCII character that terminates a delimited string.
size	A positive integer number specifying the length of a fixed length string.
sizeType	An integer data type (e.g. byte-8) which stores the value of a variable length string.
varName	A text string specifying a name that identifies this

	element in the BinX API.
--	--------------------------

- `<struct>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
dim	A positive integer number specifying the times to repeat the contents of the struct
varName	A text string specifying a name that identifies this element in the BinX API.

Component elements

<code><anyDataType></code>	A sequence of any complex or primitive data elements
----------------------------------	--

- `<union>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

Component elements

<code><discriminant></code>	Mandatory, exactly one is required.
<code><case></code>	A sequence of <code><case></code> elements specifying the base types and relevant discriminant values that are united.

- `<unsignedByte-8>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- `<unsignedInteger-32>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- `<unsignedLong-64>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.

- `<unsignedShort-16>`

Attributes

byteOrder	Either "bigEndian" or "littleEndian"
-----------	--------------------------------------

varName	A text string specifying a name that identifies this element in the BinX API.
---------	---

- `<useType>`

Attributes

typeName	Mandatory. A text string specifying the type name of the user-defined type referred to.
byteOrder	Either "bigEndian" or "littleEndian"
varName	A text string specifying a name that identifies this element in the BinX API.



Appendix B External API

B.1 Primitive data type protocol

Data Type	BinX Tag	C++ typedef	C++ native data type
8-bit signed integer	<byte-8>	bx_byte8	char
8 bit unsigned integer	<unsignedByte-8>	bx_ubyte8	unsigned char
8-bit ASCII character	<character-8>	bx_char8	char
16-bit short integer	<short-16>	bx_short16	short
32-bit integer	<integer-32>	bx_int32	long
64-bit long integer	<long-64>	bx_long64	long long
Unsigned 16-bit short integer	<unsignedShort-16>	bx_ushort16	unsigned short
Unsigned 32-bit integer	<unsignedInteger-32>	bx_uint32	unsigned long
Unsigned 64-bit long integer	<unsignedLong-64>	bx_ulong64	unsigned long long
IEEE single precision floating point number	<float-32>	bx_float32	float
IEEE double precision floating point number	<double-64>	bx_double64	double

Data Type	Data object class name	class ID constant	type comparison macro	<code>BxDataObject</code> get data member function
8-bit byte	<code>BxByte8</code>	<code>BXBYTE8</code>	<code>ISBYTE8 ()</code>	<code>getBytes ()</code>
8 bit unsigned integer	<code>BxUnsignedByte8</code>	<code>BXUBYTE8</code>	<code>ISUBYTE8 ()</code>	<code>getUnsignedByte ()</code>
8-bit ASCII character	<code>BxCharacter8</code>	<code>BXCHARACTER8</code>	<code>ISCHARACTER8 ()</code>	<code>getChar ()</code>
16-bit short integer	<code>BxShort16</code>	<code>BXSHORT16</code>	<code>ISSHORT16 ()</code>	<code>getShort ()</code>
32-bit integer	<code>BxInteger32</code>	<code>BXINTEGER32</code>	<code>ISINTEGER32 ()</code>	<code>getInt ()</code>
64-bit long integer	<code>BxLong64</code>	<code>BXLONG64</code>	<code>ISLONG64 ()</code>	<code>getLong ()</code>
Unsigned 16-bit short integer	<code>BxUnsignedShort16</code>	<code>BXUSHORT16</code>	<code>ISUSHORT16 ()</code>	<code>getUnsignedShort ()</code>
Unsigned 32-bit integer	<code>BxUnsignedInteger32</code>	<code>BXUINTEGER32</code>	<code>ISUINTEGER32 ()</code>	<code>getUnsignedInt ()</code>
Unsigned 64-bit long integer	<code>BxUnsignedLong64</code>	<code>BXULONG64</code>	<code>ISULONG64 ()</code>	<code>getUnsignedLong ()</code>
IEEE single precision floating point number	<code>BxFloat32</code>	<code>BXFLOAT32</code>	<code>ISFLOAT32 ()</code>	<code>getFloat ()</code>
IEEE double precision floating point number	<code>BxDouble64</code>	<code>BXDOUBLE64</code>	<code>ISDOUBLE64 ()</code>	<code>getDouble ()</code>

Data Type	Data object class name	constructor function
8-bit byte	BxByte8	BxByte8 (bx_byte8)
8 bit unsigned integer	BxUnsignedByte8	BxUnsignedByte8 (bx_ubyte8)
8-bit ASCII character	BxCharacter8	BxCharacter8 (bx_char8)
16-bit short integer	BxShort16	BxShort16 (bx_short16)
32-bit integer	BxInteger32	BxInteger32 (bx_int32)
64-bit long integer	BxLong64	BxLong64 (bx_long64)
Unsigned 16-bit short integer	BxUnsignedShort16	BxUnsignedShort16 (bx_ushort16)
Unsigned 32-bit integer	BxUnsignedInteger32	BxUnsignedInteger32 (bx_uint32)
Unsigned 64-bit long integer	BxUnsignedLong64	BxUnsignedLong64 (bx_ulong64)
IEEE single precision floating point number	BxFloat32	BxFloat32 (bx_float32)
IEEE double precision floating point number	BxDouble64	BxDouble64 (bx_double64)

B.2 Complex data type protocol

Data Type	BinX Tag	Data object class name	class ID constant	type comparison macro
Dataset or Structure	<dataset>	BxDataset	BXDATASET BXSTRUCT	ISDATASET() ISSTRUCT()
Union	<union>	BxUnion	BXUNION	ISUNION()
Array		BxArray	BXARRAY	ISARRAY()
Fixed size array	<arrayFixed>	BxArrayFixed	BXARRAYFIXED	ISARRAYFIXED()
Variable size array	<arrayVariable>	BxArrayVariable	BXARRAYVARIABLE	ISARRAYVARIABLE()
Array	<string>	BxString	BXSTRING	ISSTRING()
Array of characters	<string size>	BxString	BXSTRING FIXED	ISSTRING FIXED()
Array of characters	<string sizeType>	BxString	BXSTRING VARIABLE	ISSTRING VARIABLE()
Array of characters	<string delim>	BxString	BXSTRING DELIMITED	ISSTRING DELIMITED()



Appendix C Sample Programs

A number of sample application programs are provided with the BinX library. These are outlined below. For more information, see the programs and related files in the distribution.

C.1 The BinXConverter program

This program demonstrates how to parse a BinX document and read data from the associated binary file and then dump the binary data values into another binary data file with a specified byte order.

C.2 The DataBinx program

The `DataBinx` program generates a DataBinx document from a Binx document and a binary data file. In this example two files are used, which are `SchemaBinx.xml` and `BinData.bin`.

C.3 The DataBinxParser program

The `DataBinxParser` program reads a DataBinX file and produces from it:

- A binary data file containing the data in the DataBinX document
- A BinX document containing the BinX language meta-data in the DataBinX document. The BinX document refers to the binary file described above.

To run this program, type on the command line:

```
DataBinxParser <full-path>/databinx.xml <full-path>schemabinx.xml <full-path>/binary
```

C.4 The GenSchemaBinx program

The `GenSchemaBinx` program demonstrates the ability of the BinX library to produce a BinX document from in-memory data structures.

The input is a BinX document, and the output is another (differently named) BinX document rebuilt from the data structures in memory created by parsing the original BinX document.

To run the tool, type on the command line:

```
GenSchemaBinx filepath.xml outfile.xml
```

C.5 The MergeArray program

The `MergeArray` program demonstrates how to merge two arrays from two binary files into a single array in another binary file. The BinX library is used for reading from the two binary files and the output is directly saved to the new file through `stdio`. Because of the use of direct output to file, rather than using the BinX library, no BinX document describing the merged file is produced.

As the byte order conversion is done upon reading from the source files, the output data values are adjusted to the local system.

C.6 The ParseBinx program

The `ParseBinx` program demonstrates parsing a simple BinX document. It prints the data values from the binary file according to the definition in the BinX file.

C.7 The ReadHeader program

The `ReadHeader` program demonstrates parsing the header of a standard binary file using a BinX document.

Examples are included of using the program to print part of the header of a Windows `.bmp` bitmap file and also of a `.wav` digital sound file.



Appendix D DataBinX Utilities

As well as the BinX language described in the main body of this document, the BinX library supports the DataBinX language. This can be thought of as an extension to the BinX language to allow not just meta-data but also actual data to be included in a document. The DataBinX language achieves this with additional elements which are used to encode binary data (represented as character strings) rather than meta-data.

Like the BinX language, DataBinX is an application-specific dialect of XML, so there is an analogous XML schema for DataBinX which defines the elements that can be used in DataBinX documents.

The main use for DataBinX is as a BinX library internal format which allows a BinX document and its associated binary file to be woven together into a valid XML data stream containing the binary data embedded within its BinX meta-data description.

Two utilities are provided which (respectively) enable such a data stream to be generated, and separated into its original constituents:

D.1 The GenDataBinx utility

The GenDataBinx utility is used to generate a DataBinX document based on a BinX document and a binary data file.

To run the tool, type on the command line:

```
Gen databinx binx_doc.xml databinx_doc.xml
```

(It is not necessary to specify the binary file as it is given in the <dataset> element.)

D.2 The DataBinxParser utility

This is described in the previous section on the sample programs provided.



Appendix E Common mistakes

As the BinX library is written in C++, there are many pointer-related operations that may cause errors or bugs. Here are some of the obvious examples.

<pre>BxBinaryFile* pBin = new BxBinaryFile(); pBin->setFilename("filename.bin");</pre>	<p>The <code>setFilename()</code> member function doesn't copy the string, so use:</p> <pre>char* sfile = new char[N]; strcpy(sfile, "filename.bin"); setFilename(sfile);</pre> <p>or</p> <pre>copyFilename("filename.bin");</pre>
<pre>pBin->setFilename(sfile); delete[] sfile;</pre>	<p>Do not delete a string after being assigned to an object by setting.</p>
<pre>BxShort16* s = new BxShort16(100); BxDataset ds; ds.addDataObject(s); delete s;</pre>	<p>Do not delete a native data type after adding it to the data object.</p>
<pre>BxBinaryFile* p = new BxBinaryFile(); BxDataset ds; ds.setBinaryFilePtr(p);</pre>	<p>The <code>BxBinaryFile</code> instance will be not be released by the <code>BxDataset</code> object. Once the <code>BxDataset</code> instance has been deleted, execute:</p> <pre>delete p;</pre>
<pre>BxBinxFile* p = new BxBinxFile(); BxBinaryFile* pb = new BxBinaryFile(); p->setBinaryFilePtr(pb); delete pb;</pre>	<p>The <code>BxBinaryFile</code> instance will be released by the <code>BxBinxFile</code> object.</p>
<pre>printf("%s", p->toString(true));</pre>	<p>This will cause a memory leak, as the string is allocated but not released. Instead use:</p> <pre>Char* s = p->toString(true); printf("%s", s); delete[] s;</pre>
<pre>BxArray* pArray; ... BxDataObject* p = pArray->get(0);</pre>	<p>Don't forget to delete <code>p</code> after use.</p>



Appendix F Error messages

BX_ERROR_DATA	Data related error.
BX_ERROR_DATATYPE	Incompatible data type.
BX_ERROR_DATARANGE	Index not in the range.
BX_ERROR_NO_VAR	No such <code>varName</code> .
BX_ERROR_NO_TYPE	No such <code>typeName</code> .
BX_ERROR_BIN_CREATE	Binary file creation error.
BX_ERROR_BIN_NOFILE	Binary file not found.
BX_ERROR_BIN_NOTREAD	Cannot read from binary file.
BX_ERROR_BIN_BLOCKWRITE	Write error from binary file.
BX_ERROR_BIN_BLOCKREAD	Read error from binary file.
BX_ERROR_BIN	Binary file error.
BX_ERROR_DOM	BinX syntax error.
BX_ERROR_DOM_NOTBINX	Not a BinX file.
BX_ERROR_DOM_NOTYPENAME	A <code><typeDef></code> element has no attribute <code>typeName</code> .
BX_ERROR_DOM_NOTYPECONTENT	A <code><typeDef></code> element has no children.
BX_ERROR_DOM_NOTDEFINEDTYPE	Not a user-defined data type.
BX_ERROR_DOM_NOSRC	No <code>src</code> attribute in the <code><dataset></code> section.
BX_ERROR_DOM_NOFILE	No <code><dataset></code> section in BinX file.
BX_ERROR_DOM_BLOCKSIZE	Improper blocksize.
BX_ERROR_DOM_NOCHILD	No child element.
BX_ERROR_DOM_ARRAYTYPE	Array data type error.
BX_ERROR_DOM_ARRAY	Array is not used directly.
BX_ERROR_DOM_UNKNOWNTAG	Unknown tag.
BX_ERROR_EXCEPTION_FLOAT	Floating point exception.
BX_ERROR_MEMALLOC	Memory allocation failure.
BX_ERROR_BINX_OPEN	Failed to open a BinX file.
BX_ERROR_BINX_CREATE	Failed to create a BinX file.
BX_ERROR_ARRAYINDEX	Array index error.