EXIP - Embeddable EXI Processor in C

EXIP USER GUIDE

November 1, 2012

Rumen Kyusakov
PhD student, Luleå University of Technology

Contents

1 Introduction

EXIP is a free, open-source, C language implementation of the Efficient XML Interchange (EXI) standard for representing XML documents. The EXI format keeps the data in a binary form and is a compact and efficient way for storing, processing and transmitting XML structured information. Each EXI document consists of two parts: a header and a body. The header defines different encoding/decoding parameters that affect the compactness, processing efficiency and memory usage when processing the document (the EXI body part). For example, the document size can be further reduced by using the EXI compression option indicating the application of DEFLATE algorithm for data compression. Another important parameter is connected to the use of XML schema information that sets constraints on the structure and content of the document. The schema information must be available before the EXI encoding/decoding, and it should be either statically set or communicated in advance using XML schema languages such as W3C XML Schema. Schema-enabled EXI processing is much faster and results in smaller EXI documents compared to schema-less processing. Furthermore, there are a set of options, called fidelity options, determining the degree to which the EXI encoding preserves all the information items found in the XML specification. More information about EXI is given in the W3C specification [1] and primer [2]. There are other specifications/formats for increasing the compactness and efficiency of XML such as Fast Infoset, X.694 ASN.1 etc. Comparison with EXI and performance measurements are available from the W3C EXI Working Group [3, 4].

1.1 Intended audience

This user guide is structured as a step-by-step tutorial on using the EXIP library. The guide is written with the assumption that the reader has fairly good understanding of the C programming language, XML and XML Schema technologies. While the goal was to give short introductory notes on the EXI format, the reader is expected to be familiar with the description of the EXI header options provided in the specification. The intended audience of this guide are programmers working with XML/EXI technologies for applications with high efficiency requirements such as networked embedded systems, productivity tools and servers. Developers interested in contributing to the EXIP project in terms of extra features, bug reports and patches are also encouraged to read it.

This guide does not assume the use of any concrete hardware/software platform. All the examples and instructions should be valid for all execution environments.

1.2 Organization

The rest of this guide is organized in five sections. Section 2 **Basic Concepts** gives a high level overview of some important concepts for working with EXIP. Section 3 **Maturity Statement** describes the current status of the project. Sections 4 **Serialization** and 5 **Parsing** present the steps required for using the EXIP API for processing EXI streams. The last part - 7 **Schema Information** is devoted to working with XML Schema definitions used for schema-enabled EXI parsing and serialization.

1.3 Acknowledgment

The author would like to thank the European Commission and the partners of the EU FP7 project IMC-AESOP (www.imc-aesop.eu) for their support.

2 Basic Concepts

EXIP is a C library written in a portable manner that implements the EXI format for XML representation. Probably a better way of describing what is EXIP is to say what it is *not*. EXIP is not a tool for converting text XML documents to EXI and vice versa. Why is that then? To start with, the XML to EXI conversion requires a XML parser that process the XML input. The XML parser itself is at least as big chunk of code as it is the EXI parser and having them both at the same time might not be possible or desired on a resource constrained embedded system. Second, parsing the text XML and then converting it to EXI effectively removes all the processing benefits of EXI. Having said that, it does not mean that it is not possible to use EXIP in such scenario. For example, it is planned as a future work to include a module in EXIP that performs exactly that: generating corresponding EXI streams from a text XML input and vice versa. It is therefore an optional behavior and not an only possible way of using the library. It is not even difficult to implement such a module and the reader could do that as a practical exercise after going through this guide.

As a result of this design choice, EXIP cannot use (at least for now) the XML Schema format directly to perform schema-enabled processing. This might sound as a big flaw in the implementation but is just the opposite. XML Schema documents are plain XML documents and as such they have analogous EXI representation. Working with the EXI representation of the XML Schema definitions brings all the performance benefits of the EXI itself - faster processing and more compact representation. Now, using static systems where the schema information is only processed at design time would not make much difference. However, once you are faced with more dynamic systems that are capable of handling schema information at run-time, the use of EXI representation is more beneficial especially in networked embedded environments.

Yet another *not* - EXIP is not compliant with DOM, SAX or StAX Application Programming Interfaces (APIs) for XML processing. The single reason for that is the efficiency trade-off. All these APIs are using string representation of the primitive data types defined by the XML Schema specification such as float, integer, date etc. This means that when schema definitions are available these types must be converted from native types to string and then back from string to native representation in order to fit in the API. Once again this does not mean that you cannot use EXIP with applications that require DOM, SAX or StAX interface. The EXIP API is low level and typed and requires a wrapper module in order to provide the aforementioned interfaces, which again is scheduled for future work.

Figure 2.1 depicts these design decisions and shows the different components of the library. Key concept when creating the structure of the project was modularity - the functionality is grouped and encapsulated in different components. This allows for disabling features that are not needed directly at compile time. As an example, the Schema Parser component that is responsible for generation the grammar structures based on XML Schema definitions is only needed if the system is expected to dynamically

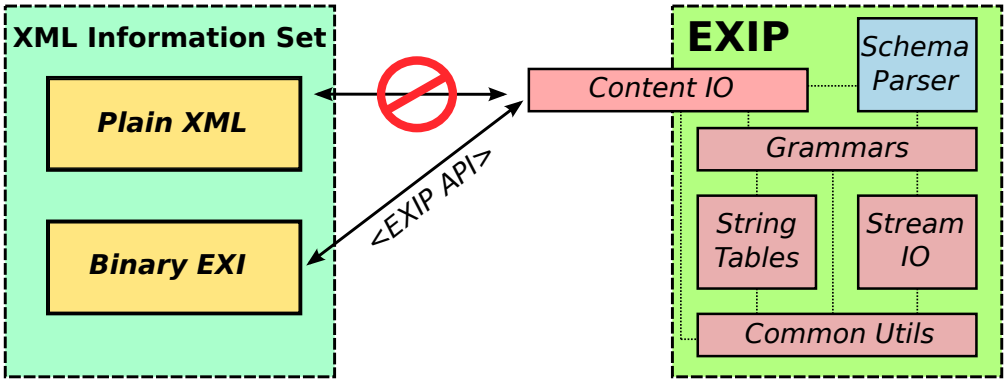

Figure 2.1: *EXIP components*

handle XML schemes and hence in all other cases can be removed from the build at compile time.

For further information and details on the EXIP API and the rational behind it see the paper that first presents EXIP [5] (please refer to this work when citing information from this guide or other EXIP documentation).

2.1 Project structure

The EXIP library is an Eclipse C project (and MS Visual Studio 2010 solution) structured in the following subdirectories:

- `bin/` - this is the build output directory. It is automatically created/removed when doing builds of the project. The library binaries are located directly into the `bin/` folder while the examples and other utils are compiled in separate subfolders

- `build/` - this folder contains the build scripts of the project. The current release provides a MS VS 2010 build under `build/vs2010/` and *GCC* toolchain builds under `build/gcc/`. The `build/gcc/` folder further contains platform-dependent parameters and headers located in subfolders - `build/gcc/mulle` for the Mulle sensor platform, `build/gcc/oe-armv5te` for OpenEmbedded Linux on ARMv5te and the default `build/gcc/pc` for descktop PCs. The build is started from `build/gcc/` with the following `make` build targets available:

  all compiles the EXIP library. The default target platform is PC. To change the target platform to Mulle for example, add `target=mulle` as an argument of `make`

  check invokes the Check unit tests

  clean removes the `bin/` folder

  doc generates the project Doxygen documentation. The output directory is set to `doc/dev/doxygen/`

  examples compiles and builds the example executables

  utils compiles and builds the utils executables

  lib generates a static library libexip.a in `bin/lib`

dynlib generates a dynamic library libexip.so in `bin/lib`

- `doc/` - the project documentation: `doc/user/` contains the source of this user guide; `doc/dev/` is the development documentation; `doc/www/` contains the information available on the project web page
- `examples/` - the root of the example applications shipped with EXIP
- `include/` - this folder contains the public header files of the EXIP library. Together with the `exipConfig.h` defined per target platform, these files define the public interface of the library.
- `src/` - here are all internal definitions and `.c` files divided into subfolders according to the modules in which they belong
- `tests/` - Check unit tests, integration and validation tests
- `utils/` - supporting utilities and development tools

2.2 Strings in EXIP

All character strings in EXIP are length prefixed. This means that all the strings that are passed to the EXI encoder and all the strings that are parsed from the decoder are in this format. The motivation for that is the number of string comparisons that the EXI processor must perform when going through an EXI stream. In many cases, when the strings differ in length, this operation is very efficient when the length is saved together with the characters data. EXIP provides an API for working with this type of strings defined in `stringManipulate.h`. The implementation of the functions declared in this header file defines what type of characters are supported in the string. For example, `ASCII_stringManipulate.c` implements the strings as ASCII character arrays.

One way of increasing the compactness defined in the EXI specification is through the use of string tables. The string tables are partitioned dictionary that stores and indexes the strings that occur in particular EXI document. In such way, when a string has been used once in the document it is added to the string tables. Every other occurrence of the same string in the document is represented by its index in the string tables.

Figure 2.2 shows how the string tables are implemented in EXIP. As a user of the library you do not need to understand the actual implementation details. The only thing that you need to be aware of is that the strings that are passed to your application by the EXIP parser should not be modified as they are integral part of the string tables. You can clone the strings and use that copy for string manipulations.

2.3 Error handling and memory management

When an invalid input is given to the EXIP parser or some other error conditions occur, the EXIP library functions return a numeric error codes that are defined in `errorHandle.h`. More fine-grained error messages can be acquired by turning on the debugging routines. You have control over the level of verbosity (INFO, WARNING or ERROR) and the source of debugging information. All these parameters can be configured in the `exipConfig.h` header that is defined per target platform

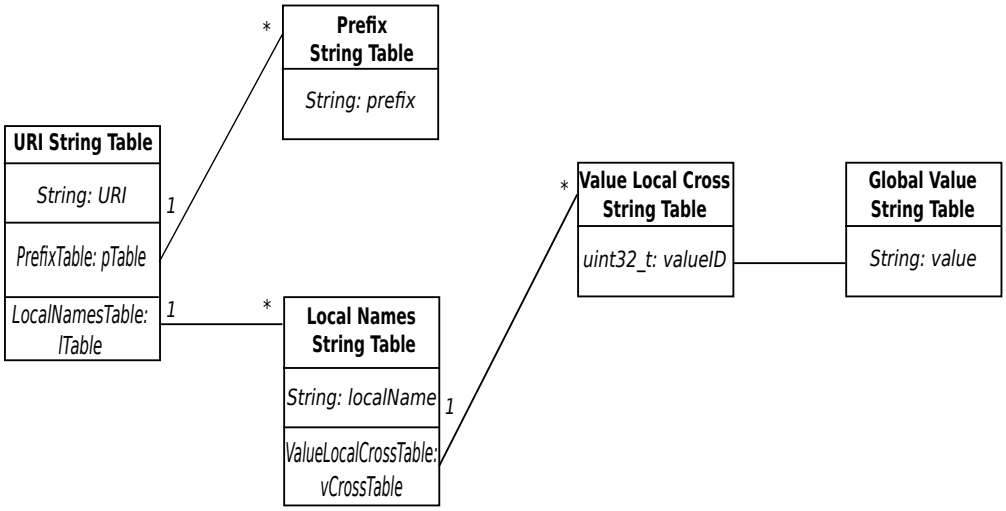

Figure 2.2: *String tables in EXIP*

in `build/gcc/<target_platform>` or `build/vs2010` for Windows. When turned on the debugging information is by default printed on the standard output.

EXIP has an internal memory management infrastructure and you are not required to know its details unless you are using the EXIP string type in your own applications. That is because the string manipulation functions use `AllocList` to manage their allocations. You can either define your own string manipulation functions or acquaint yourself with the memory management routines defined in `memManagement.h`.

3 Maturity Statement

The latest release of the EXIP library is alpha version 0.4. As such, some features are missing and others are not stable enough for real-world usage. Here is the status of the EXIP functionality:

- Currently only ASCII strings are supported. It should be relatively easy to extend this with Unicode strings by implementing a dozen of functions in the header file `stringManipulate.h`
- Using default options in the EXI header and schema-less mode should work without any issues
- Schema-enabled processing can cause problems with some XML schema constructs. The reason is in the grammar generation utility that converts XML Schema definitions to EXIP grammars. It still does not cover all the features of the XML Schema specification such as string pattern facets, unions and lists.
- The following EXI header options are not supported yet: byte-alignment, pre-compression, compression, preserving lexical values, random access (selfContained), datatype representation map.

4 Serialization

This section describes the process of serializing an EXI stream using the EXIP API. The description is divided into two parts - schema-less and schema-enabled encoding. The basic steps are essentially the same in both cases so the schema-enabled part shows only some special use-cases and configurations that differ from the schema-less serialization.

As discussed in section 2, the EXIP API is especially useful for applications that are not dependent on XML APIs and text XML inputs. Example use cases are embedded devices working exclusively with EXI, XML binding tools that generate parsing/serialization code from XML Schema definitions, productivity tools etc. If your application works with text XML input that needs to be converted to EXI you need to create an external module that is built on top of the EXIP serialization API.

The serialization API is defined in `EXISerializer.h`. You can work with the serialization functions directly or use the `serialize` global variable to access these functions as methods.

4.1 Schema-less encoding

The serialization of an EXI stream consists of 7 simple steps:

1. Declare a stream container that holds the serialized data and EXIP state:

```
EXIStream strm;
```

2. Initialize the EXI header of the new stream container:

```
serialize.initHeader(&strm);
```

3. (Optional) Set any options in the EXI header, if different from the defaults:

```
strm.header.has_options = TRUE; /* Set the options presence bit */
strm.header.has_cookie = TRUE; /* Include EXI cookie in the header */
SET_STRICT(strm.header.opts.enumOpt); /* Indicate a STRICT mode encoding */
```

   (see Section 6 for more information on how to set up the EXI options)

4. Define a binary output buffer for storing the serialized EXI data. The buffer can optionally include an external stream for sending the serialized output (flushing the buffer when full). If an output stream is not defined the whole EXI document must fit into the memory buffer. The following code declares and initializes an output memory buffer:

```
BinaryBuffer buffer;
char buf[OUTPUT_BUFFER_SIZE];

buffer.buf = buf;
buffer.bufLen = OUTPUT_BUFFER_SIZE;
buffer.bufContent = 0;
```

   Beside these definitions, when a file is used to store the serialized data the following steps are also required:

```
/* IN CASE OF FILE OUTPUT STREAM */
/* Define a function implementing the actual writing to a file */
size_t writeToFileOutputStream(void* buf, size_t writeSize, void* stream)
{
  FILE *outfile = (FILE*) stream;
```

```
  return fwrite(buf, 1, writeSize, outfile);
}

FILE *outfile; /* Using a file for storing the serialized data */
outfile = fopen(destinationEXIFile, "wb"); /* open the file before use */
buffer.ioStrm.readWriteToStream = writeToFileOutputStream;
buffer.ioStrm.stream = outfile; /* Sets the output stream to the file */
```

If no file or other output streams are being used (e.i. the whole EXI message is stored in `buffer.buf`), the output function and the stream of the `BinaryBuffer` must be initialized with NULL values:

```
/* IN CASE OF IN-MEMORY ONLY OUTPUT*/
buffer.ioStrm.readWriteToStream = NULL;
buffer.ioStrm.stream = NULL;
```

The size of the `OUTPUT_BUFFER_SIZE` macro depends on the use case. When in in-memory output mode or sufficient RAM on the platform the buffer size should be kept high. Note that even if an output stream is used such as a file, a non-zero length buffer for temporary storing parts of the EXI serialization is still needed.

5. Initialize the stream:

```
/**
 * @param[in, out] strm EXI stream container
 * @param[in, out] buffer output BinaryBuffer for storing the encodded EXI stream
 * @param[in] NULL a compiled schema information to be used for schema
          enabled processing; NULL if no schema is available
 * @param[in] SCHEMA_ID_ABSENT one of SCHEMA_ID_ABSENT, SCHEMA_ID_SET,
    SCHEMA_ID_NIL or SCHEMA_ID_EMPTY as defined in the specification
 * @param[in] NULL when in SCHEMA_ID_SET mode a valid string representing
          the schemaID; NULL otherwise
 */
serialize.initStream(&strm, buffer, NULL, SCHEMA_ID_ABSENT, NULL);
```

6. Start building the stream step by step starting from the header. The body should always start with `startDocument()` and ends with `endDocument()` with at least one element declaration. In schema-less mode the only type of data allowed is `string` so when encoding element or attribute values you should use `stringData()`. **NOTE:** different attributes within a single element should be encoded in a lexicographical order during serialization!

```
/* Static String type definitions if any */
const String NS_TARGET_STR = {"http://www.ltu.se/exip", 22};
const String NS_EMPTY_STR = {NULL, 0};
const String ELEM_ROOT_STR = {"rootElement", 11};
const String ATTR_TEST_STR = {"testAttribute", 13};

serialize.exiHeader(&strm); /* Always first */
serialize.startDocument(&strm); /* Indicates beginning of the EXI body */

QName qname= {&uri, &ln, NULL}; /* QName definition cosisting of
namespace, local name and prefix if any*/

/* Encode an element with QName <http://www.ltu.se/exip:rootElement> */
qname.uri = &NS_TARGET_STR;
qname.localName = &ELEM_ROOT_STR;
serialize.startElement(&strm, qname);

/* Encode attribute with QName <testAttribute> and indicate its type */
qname.uri = &NS_EMPTY_STR;
qname.localName = &ATTR_TEST_STR;
```

```
serialize.attribute(&strm, qname, VALUE_TYPE_STRING);

String ch; /* EXIP string representing a string value */

/* Convert a static ASCII string constant to EXIP string type */
asciiToString("attribute value", &ch, &strm.memList, FALSE);

serialize.stringData(&strm, ch); /* Encode the value of the attribute */

asciiToString("element value", &ch, &strm.memList, FALSE);

serialize.stringData(&strm, ch); /* The value of the <rootElement> */

serialize.endElement(&strm); /* Close element <rootElement> */

serialize.endDocument(&strm); /* Close the EXI body*/
```

The above code produces an EXI stream that has the following XML representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:rootElement xmlns:xp="http://www.ltu.se/exip"
    testAttribute="attribute value">
element value
</xp:rootElement>
```

7. Destroy the EXI stream container and free the memory allocated by it. If any other streams are left open close them as well:

```
serialize.closeEXIStream(&strm);
fclose(outfile); /* Close the file (if any) used to store the output data */
```

4.2 Schema-enabled encoding

When in schema-enabled mode the basic serialization steps are essentially the same. The difference is in the parameters passed to the `serialize.initStream()`. You need to pass a valid `EXIPSchema` object and not NULL as in the schema-less case. This object contains all the definitions and constrains from the XML schema and its creation is a topic of section 7 **Schema Information**. During the encoding in Step 6 you must use the types for the values as defined by the schema definitions. For example, `serialize.intData()` for integer values of elements or attributes and `serialize.booleanData()` for boolean values. There are also few options in the EXI header that influence how the schema information is utilized when encoding the EXI body:

strict this is a boolean option that specify if deviations from the schema definitions are allowed (has value of `false`) or are not allowed (has value of `true`) in the EXI body. The default value of this option is `false`. When in STRICT mode the representation is slightly more compact. You can set the value of this option in Step 3 with the following macro:

```
SET_STRICT(strm.header.opts.enumOpt); /* Indicate a STRICT mode encoding */
```

schemaId this option takes a string for its value. There are four possible states for this option as defined in the specification:

- SCHEMA_ID_ABSENT - default state; no statement is made about the schema information. If it is a schema-less or schema-enabled mode and what schema definitions to be used when decoding must be communicated out of band.

- SCHEMA_ID_SET - a string identification of the schema used for encoding is specified in the EXI header. The format of this schema identifier is not standardized and is left for the application to decide what to use. One possibility is to use the target namespace of the schema or the URL address of the schema location.
- SCHEMA_ID_NIL - no schema information is used for processing the EXI body (i.e. a schema-less EXI stream).
- SCHEMA_ID_EMPTY - no user defined schema information is used for processing the EXI body; however, the built-in XML schema types are available for use in the EXI body through the `xsi:type` attribute.

All of these states can be set during initialization - Step 5:

```
String schemaID;
asciiToString("http://www.ltu.se/schema", &schemaID, &strm.memList, FALSE);
serialize.initStream(&strm, buffer, OUTPUT_BUFFER_SIZE, &out, NULL,
        SCHEMA_ID_SET, schemaID);
```

In this example we defined the state to SCHEMA_ID_SET and set an identifier for the schema used - `http://www.ltu.se/schema`.

The serialization interface defined in `EXISerializer.h` has a more efficient low level API for encoding the document structure when in schema-enabled mode. This API can be accessed through `serializeEvent()` function. A description on how it can be used might be included in the future versions of this guide. If you want to know more about it you can check how it is used to encode the EXI options document in the header of an EXI stream in `headerEncode.c`.

5 Parsing

The parsing interface of EXIP is similar to SAX and StAX. The main difference is that most of the XML schema build-in types are passed in a native binary form rather than as a string representation. The EXIP parser processes the EXI stream in one direction and produces events on occurrence of information items such as elements, attributes or content values. The applications using EXIP must register callback functions for the events that they are interested in. In this way the data from the EXI stream is delivered to the application as a parameters to these callback functions. The header `contentHandler.h` provides declarations of the callback functions while the API for controlling the progress through the EXI stream is in `EXIParser.h`.

5.1 Schema-less decoding

When in schema-less mode all value items are encoded with strings and hence delivered to the application through the `stringData()` callback handler. Similarly to the serialization, the parsing consists of 7 simple steps:

1. Declare a parser container that holds the serialized data and EXIP state:

```
Parser parser;
```

2. Define a binary input buffer for reading the EXI stream. The buffer can optionally include an external input stream for fetching the serialized EXI data. If an input stream is not defined the whole EXI document must be stored in memory before starting the parsing process. The following code declares and initializes an input memory buffer:

```
BinaryBuffer buffer;
char buf[INPUT_BUFFER_SIZE];

buffer.buf = buf;
buffer.bufLen = INPUT_BUFFER_SIZE;
buffer.bufContent = 0;
```

Beside these definitions, when a file is used to fetch the EXI data the following steps are also required:

```
/* IN CASE OF FILE INPUT STREAM */
/* Define a function implementing the actual reading from a file */
size_t readFromFileInputStream(void* buf, size_t readSize, void* stream)
{
  FILE *infile = (FILE*) stream;
  return fread(buf, 1, readSize, infile);
}

FILE *infile; /* Using a file for fetching the EXI data */
infile = fopen(sourceEXIFile, "rb" ); /* open the file before use */
buffer.ioStrm.readWriteToStream = readFromFileInputStream;
buffer.ioStrm.stream = infile; /* Sets the input stream to the file */
```

If no file or other input streams are being used (e.i. the whole EXI message is stored in `buffer.buf`), the input function and the stream of the `BinaryBuffer` must be initialized with NULL values:

```
/* IN CASE OF IN-MEMORY ONLY INPUT*/
buffer.ioStrm.readWriteToStream = NULL;
buffer.ioStrm.stream = NULL;
```

The size of the `INPUT_BUFFER_SIZE` macro depends on the use case. When in in-memory input mode or sufficient RAM on the platform the buffer size should be kept high. Note that even if an input stream is used such as a file, a non-zero length buffer for temporary storing parts of the EXI data is still needed.

3. Define application data and initialize the parser object:

```
/** The application data that is passed to the callback handlers*/
struct applicationData
{
  unsigned int elementCount;
  unsigned int nestingLevel;
} appData;

/**
 * @param[in, out] parser EXIP parser container
 * @param[in] buffer binary buffer for fetching EXI encoded data
 * @param[in] NULL a compiled schema information to be used for schema
          enabled processing; NULL if no schema is available
 * @param[in, out] appData application data to be passed to the callback handlers
 */
initParser(&parser, buffer, NULL, &appData);
```

4. Initialize the application data and register the callback handlers with the parser object:

```
appData.elementCount = 0; /* Example: the number of elements passed */
appData.nestingLevel = 0; /* Example: the nesting level */

parser.handler.fatalError = sample_fatalError;
parser.handler.error = sample_fatalError;
parser.handler.startDocument = sample_startDocument;
parser.handler.endDocument = sample_endDocument;
parser.handler.startElement = sample_startElement;
parser.handler.attribute = sample_attribute;
parser.handler.stringData = sample_stringData;
parser.handler.endElement = sample_endElement;

/** According to the above definitions:
  *  When the parser start parsing the body, the sample_startDocument()
  *  callback will be invoked; when a start of an element is parsed the
  *  sample_startElement() will be invoked and so on. All of the events
  *  for which there is no handler registered will be discarded.
  */
```

NOTE: If the EXI options are communicated with an out-of-band mechanism and are not included in the EXI header of the stream they must be set here (see Section 6 for more information on how to set up the EXI options).

5. Parse the header of the stream:

```
parseHeader(&parser);
/* The header fields are stored in parser.strm.header*/
```

6. Parse the body of the EXI stream, one content item at a time:

```
errorCode error_code = ERR_OK;
while(error_code == ERR_OK)
{
  error_code = parseNext(&parser);
}

/**
 * On successful parsing step, the parseNext() returns ERR_OK if there
 * are more content items left for parsing and PARSING_COMPLETE in case
 * the parsing is complete. If error conditions occur during the
 * process it returns an error code.
 */
```

7. Destroy the parser object and free the memory allocated by it. If any other streams are left open close them as well:

```
destroyParser(&parser);
fclose(infile);
```

5.2 Schema-enabled decoding

When in schema-enabled mode the basic parsing steps are essentially the same. The difference is in the `EXIPSchema*` parameter passed to the `initParser()`. You need to pass a valid `EXIPSchema` object and not NULL as in the schema-less case. This object contains all the definitions and constrains from the XML schema and its creation is a topic of section 7 **Schema Information**. During parsing, the value types as defined by the schema definitions are delivered through the corresponding callback handlers and not only `stringData()`. For example, `xsd:integer` is delivered through `intData()` and `xsd:boolean` through `booleanData()`. Apart from that all other decoding steps are the same as in schema-less mode.

6 EXI Options

The EXI specification defines a set of options that can be used to alter the way the EXI body is processed. The precise values of the EXI options must be known by the EXI processors to correctly interpret the EXI streams. One way to communicate the options used for processing is to include them in the EXI Header (Recommended). However, the presence of the EXI Options in its entirety in the EXI header is optional according to the EXI specification. This means that if the options are not included in the EXI header they must be communicated with an out-of-band mechanism.

During encoding, the options are included in the EXI Header if the `has_options` flag is set to TRUE:

```
strm.header.has_options = TRUE;
```

If the flag is not set to TRUE, the specified options are used for encoding and must be known (communicated out-of-band) by the decoder as they are not included in the header.

During decoding, if the options are not present in the EXI Header they must be known and set before the parsing of the header with `parseHeader()`. If out-of-band options are set for decoding and there are options specified in the EXI stream - the out-of-band options will be ignored. If there is no value specified for a particular option in the EXI Header or by out-of-band mechanism then the default value for that option is assumed according to the EXI specifications.

Table 6.1 shows how to set the Header fields and each of the EXI options before processing EXI stream with EXIP.

EXI Option / Header field	**Instructions**
EXI Cookie	`strm.header.has_cookie = TRUE;` (Default is FALSE)
Options Presence Bit	`strm.header.has_options = TRUE;` (Default is FALSE)
EXI Version	`strm.header.version_number = 2;` (Default is 1)
alignment	`SET_ALIGNMENT(strm.header.opts.enumOpt, V);` (Where V in {`BIT_PACKED, BYTE_ALIGNMENT, PRE_COMPRESSION`})
compression	`SET_COMPRESSION(strm.header.opts.enumOpt);`
strict	`SET_STRICT(strm.header.opts.enumOpt);`
fragment	`SET_FRAGMENT(strm.header.opts.enumOpt);`
preserve	`SET_PRESERVED(strm.header.opts.preserve, V);` (Where V in {`PRESERVE_COMMENTS, PRESERVE_PIS, PRESERVE_DTD, PRESERVE_PREFIXES, PRESERVE_LEXVALUES`})
selfContained	`SET_SELF_CONTAINED(strm.header.opts.enumOpt);`
schemaId	`const String SCHEMA_ID_STR = {"http://www.ltu.se/exip", 22}; strm.header.opts.schemaID = SCHEMA_ID_STR;`
datatypeRepresentationMap	Not implemented yet!
blockSize	`strm.header.opts.blockSize = 1000;`
valueMaxLength	`strm.header.opts.valueMaxLength = 100;`
valuePartitionCapacity	`strm.header.opts.valuePartitionCapacity = 100;`
[user defined meta-data]	Not implemented yet!

Table 6.1: *Instructions on how to set each EXI options before processing*

7 Schema Information

This section examines different ways of constructing `EXIPSchema` object containing the XML schema definitions and constrains. The header file `grammarGenerator.h` defines a function `generateSchemaInformedGrammars()` that takes a XML schema document(s) and converts them to `EXIPSchema` object. This function can be used to dynamically (at run-time) parse a schema and generate the EXI grammar constructs for schema-enabled parsing and serialization. The `EXIPSchema` object can be used to process multiple EXI streams. After the processing is done, the schema object can be destroyed to free the allocated memory with the `destroySchema()` function.

As mentioned earlier, the EXIP library currently supports only XML schema definitions represented in EXI format. Moreover, the fidelity option `Preserve.prefixes` must be set in order to decode the QNames in the value items correctly (see the EXI specification for more information on that).

When the XML schemes are static (only used at compile time) the `grammarGen` module is not needed and can be excluded from the build. In this case the `EXIPSchema` object can be build from automatically generated source code by the `exipg` utility implemented in `utils/schemaHandling/createGrammars.c`.

7.1 Using exipg utility

The `exipg` utility is a command line tool for generating EXI grammar definitions for schema-enabled EXI processing based on XML schemes. It uses the grammar generation function in `grammarGenerator.h` and as such also requires EXI encoded XML schemes. There are three modes defining the output of the tool:

exip In this mode the XML schema is serialized into an EXIP specific format. The output is an EXI document that later can be loaded into the EXIP library for schema-enabled processing. This option is currently not implemented.

text In this mode the grammar definitions are printed in human readable form. This mode is useful for debugging purposes.

src In this mode the grammar definitions are generated in `C` source code. The code can use either static definitions (everything is in the programming memory) or dynamic definitions (dynamic memory allocations). The dynamic definitions are not supported in this release.

As an example, the command line arguments used to generate the EXI Options document grammars are:

```
exipg -src=static -pfx=ops_ -mask=1000000 EXIOptions-xsd.exi staticEXIOptions.c
```

7.2 Converting XML Schema files to EXI

Currently, there are no XML Schema editing tools that are capable of saving the document in EXI format. For that reason it is required that you convert the text XML Schema to EXI encoding before using it with EXIP. You can use any of the open source Java implementations of EXI for that purpose. Particularly convenient for use is the ExiProcessor command line utility. The correct arguments to be passed to the utility for converting the EXIOptions.xsd are:

```
java -jar ExiProcessor.jar -header_options -preserve_prefixes
  -xml_in EXIOptions.xsd -exi_out EXIOptions-xsd.exi
```

References

[1] *Efficient XML Interchange (EXI) Format 1.0*, W3C Std., March 2011. [Online]. Available: http://www.w3.org/TR/2011/REC-exi-20110310/

[2] D. Peintner and S. Pericas-Geertsen, "Efficient XML Interchange (EXI) Primer," W3C, Tech. Rep., 2009. [Online]. Available: http://www.w3.org/TR/2009/WD-exi-primer-20091208/

[3] C. Bournez, "Efficient XML Interchange Evaluation," W3C, Tech. Rep., 2009. [Online]. Available: http://www.w3.org/TR/exi-evaluation/

[4] G. White, J. Kangasharju, D. Brutzman, and S. Williams, "Efficient XML Interchange Measurements Note," W3C, Tech. Rep., 2007. [Online]. Available: http://www.w3.org/TR/exi-measurements/

[5] R. Kyusakov, J. Eliasson, and J. Delsing, "Efficient Structured Data Processing for Web Service Enabled Shop Floor Devices," in *20th IEEE International Symposium on Industrial Electronics*, 2011.