Introduction

The purpose of this document is to identify and prioritize issues in the current version of the EXIP library and to suggest a solution that comply with the vision of all developers.

Problem 1: Scope issue

SOLVED

Description

Elements with the same QName can reside in a different scope:

- In the global scope
- Nested in some complex type definition

These elements are essentially distinct units and may have different types. As such, to uniquely identify the grammar for a particular element you need:

- The scope
- The element QName

The scope of a complex type definition can be in the scope of another definition i.e. the scope can be nested. For example, in the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="TestNS"
xmlns:tns="TestNS" elementFormDefault="qualified">

<element name="a" type="integer"></element>
<complexType name="a">
    <sequence>
        <element name="b" type="float"></element>
        <element name="c">
            <complexType>
                <sequence>
                    <element name="b" type="boolean"></element>
                </sequence>
            </complexType>
        </element>
    </sequence>
</complexType>
<element name="elem">
    <complexType>
        <sequence>
            <element name="b" type="tns:a"></element>
            <element name="c">
                <complexType>
                    <sequence>
                        <element name="b" type="string"></element>
                    </sequence>
                </complexType>
            </element>
        </sequence>
    </complexType>
</element>
<element name="b" type="boolean"></element>
</schema>
```

The element `<element name="b" type="tns:a"/>` has nested scope `<TestNS:elem>` as it is defined within the anonymous complex type of `<elem>`. On the other hand, `<element name="b" type="string"/>` has nested scope of `<TestNS:elem>, <TestNS:c>`.

Robert[1]: I use the term "nested scope" as "scope" on its own is really just local.

Rumen[1]: yes scope is always local. I used "nested scope" instead just to make this clear as in the XSD spec the {scope} is defined simply as a complex type, which confused me at the beginning.

To illustrate all the complexity of the scope issue, consider the following XML instance document that is valid XML according to the above schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<elem xmlns="TestNS">
  <b>
    <b>0.1</b>
    <c>
      <b>true</b>
    </c>
  </b>
  <c>
    <b>string-type</b>
  </c>
</elem>
```

Here `<b>` has 4 different types, hence 4 different grammars with the same QName but with different nested scope.

One more thing to consider is that in the global scope you can have elements and types with the same QName and different grammars. In the schema above, there is:

- `<element name="a" type="integer"></element>`
- `<complexType name="a">`

This means that the `LnEntry` in the string tables should support two links to EXIGrammars - one for an element and one for a type. The built-in element grammars are also defined within the global scope and the global element grammar link in `LnEntry` can safely be reused for that purpose.

The main problem with the current GRAMMAR_TABLE solution is that it assumes all elements have named types (either complex or simple) and thus these types can be found in the global scope of the string tables. However, there are also elements with anonymous types as the element "compression" below:

```
<xsd:element name="common" minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="compression" minOccurs="0">
              <xsd:complexType />
                ......
          </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Another issue is that the `GrammarTable` in the `LnEntry` is a bit redundant because for a particular QName in global scope you can have at most 2 EXIGrammars - one for an element and one for a type. So this table can have at most 2 elements.

Robert[1]: Maybe I am missing something here but the `LnTable` contains entries for all QNames, not just those in global scope. For example, in the SEP 2.0 schema, "value" has 8 entries in GrammarTable as it is declared with 8 different types in various scopes. Therefore, as to what you are saying above, that QName may appear in many different nested scopes. At the moment, each grammar entry is qualified with a QNameID which identifies its type. I guess the problem is that whilst it does take into account the repetition of the QName in different scopes with different types, it doesn't automatically take into account the nesting as the document is encoded. One simple but clumsy solution would be to manually identify the scope when the QName is used in the encoding. This may not be as bad as it sounds but is clearly clumsy.

Rumen[1]: I think I did not fully understand the idea of `GrammarTable` and `LocalTable`. If you store all 8 occurrences in one `GrammarTable` (the one with `LnEntry` http://zigbee.org/sep:value) and the `qname` being the qname of the actual type such as Int48, Int16, String42 etc. then what is the role of `LocalTable`? Also what if you have a ninth occurrence of value in some scope that uses an anonymous type:

```
<xsd:element name="value">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="subValue" minOccurs="0">
              <xsd:complexType />
                ......
          </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

The grammar for this type does not have a QNameID beside the http://zigbee.org/sep:value and in the `GrammarTable` the

`qname` field will be empty.
In any case, having two tables per `LnEntry` is huge overhead. Both size-wise and performance-wise when you need to look in the table to find the correct grammar. The idea to use the grammar production is very optimal as the SE(QNAME) productions themselves contain the scope information.

Robert[2]: I think we should go with your approach and ignore what I have done to a large extent with GrammarTable and LocalTable. I admit it was done in a rush to get things working well enough for a test event and has not had enough thought. In other words, I have no objection at all to removing the concepts and we should look at it from a fresh approach.

The current `LocalTable` in `LnEntry` cannot account for the nested scope described above. The `LocalTable` in the `LnEntry` is only needed if an element with the QName defined by the `LnEntry` is available in the schema and the type of this element is a complex type - either named or anonymous. In case of an anonymous type the scope is unique. In case of a named complex type the scope of all elements that has this type will be the same and hence needed to be created physically only once.

Suggestion

The simplest approach and the most efficient one is to add a link to the correct grammar in the grammar productions:

```
struct Production
{
    EventType eventType; // (enum EVENT_SD, EVENT_SE_QNAME, EVENT_AT_ALL etc.)
    QNameID qname;
    /**
     * type is an index in an array of Grammars in the EXIP schema when the event
     * is SE(qname) (Requires the solution of Problem 2)
     * In case of CH or AT event the type is an Index in the SimpleTypeTable
     * that contains detailed information about the type including the EXI data type
     */
    Index type;
    Index nonTermID; // unique identifier of right-hand side Non-terminal
};
```

Then the `LnEntry` will look like this:

```
struct LnEntry {
    VxTable vxTable;
    String lnStr;

    /** Stores a pointer to global element grammars only */
    EXIGrammar* elemGrammar;

    /** Stores a pointer to global type grammars only - either complex or simple */
    EXIGrammar* typeGrammar;
};
```

`elemGrammar` will be `NULL` if there is no global element with that qname. This pointer is also used for built-in element grammars in schema-less and mixed processing. To be changed to Index and INDEX_MAX (see Problem 2)

`typeGrammar` will be `NULL` if there is no global type grammar with that qname. To be changed to Index and INDEX_MAX (see Problem 2)

NOTE: Most probably global attribute definitions should also be stored in the global scope. This can be done by implementing a sorted array of global attribute definitions in the EXIPSchema object. This will rarely be used for lookups during processing so that solution should work in a decent way. This can be left as a future work as global attribute definitions are not very common any way.

Problem 2: Problems with handling of the grammar serialization

SOLVED

Description

Currently, the EXI grammars are not stored in a centralized location within the EXIPSchema object but instead

multiple pointers spread in the LnEntries in the string tables are pointing to them. This leads to two issues:

1. During serialization (exipg code) you need to make sure a single grammar is not serialized more than once. Currently, a hashtables with the grammars are used to check for that which makes the code very messy and bug prone.
2. During grammar augmentation you need to walk through all the string table entries even though only small part of the LnEntries contain unique EXI grammar.

This problem will be a number of times more severe if we implement the suggested solution to Problem 1 as there will be nested tables with local scope and even more pointer links to EXI grammars.

Suggestion

Store all the grammars in the EXIPSchema object in an array and use indexes of the array to identify a particular grammar in the LnEntry of the string tables and not pointers. This will make the serialization much more straight forward. In case, the simple types and the EXI grammars are represented with the same structure (see Problem 3), only one grammar table is needed in EXIPSchema and SimpleTypeTable will contain all the grammars not only the simple types.

NOTE: This problem should be solved before Problem 1 in order to save the efforts to modify the messy code in exipg.

Robert[1]: This is a really good idea and I agree this should be done first

Rumen[1]: I am working on that currently. I'll try to have a version of that more or less running today. If you agree to change the current version of the scope issue - GRAMMAR_TABLE with adding the scope information to the productions I can merge the branch to the trunk - all the grammar tables stuff is missing there? Think about it again and lets decide what will do for that.

Problem 3: Simple type grammars are redundant

Description

Simple type grammars always have the same structure:

```
NT-0:
    CH [<THE ACTUAL TYPE>]    NT-1    0

NT-1:
    EE 0
```

So we don't need to store them separately. This will save a lot of space and RAM especially in STRICT FALSE mode as each of this grammars after augmentation has the following structure:

```
NT-0:
    CH [<THE ACTUAL TYPE>]                                          NT-1  0
    EE                                                                    1.0
    AT ([2:1]http://www.w3.org/2001/XMLSchema-instance:type) [qname] NT-0  1.1
    AT ([2:0]http://www.w3.org/2001/XMLSchema-instance:nil) [bool]   NT-0  1.2
    AT (*) [N/A]                                                    NT-0  1.3
    SE (*)                                                          NT-2  1.4
    CH [untyped]                                                    NT-2  1.5
    AT (*) [untyped]                                                NT-0  1.6.0

NT-1:
    EE                                                              0
    SE (*)                                                          NT-1  1.0
    CH [untyped]                                                    NT-1  1.1

NT-2:
    CH [<THE ACTUAL TYPE>]                                          NT-1  0
    EE                                                                    1.0
    SE (*)                                                          NT-2  1.1
    CH [untyped]                                                    NT-2  1.2
```

The only thing that we need to store is <THE ACTUAL TYPE> which includes the EXI encoding type along with any restrictions defined in the schema. struct ValueType contains the EXI encoding type and an index to the simple type table, which in turn indicates the restrictions, so this will work. So instead of creating a number of almost identical simple type grammars (one for each simple type definition) we can create just one and simply

set the <THE ACTUAL TYPE> when processing.

Suggestion

So, we only need to store one simple type grammar before augmenting and augment the grammar after the parsing of the EXI header. Then at runtime adjust the <THE ACTUAL TYPE> of the simple type grammar to match the simple type definitions.

One big change that needs to be implemented in order for that to work is to change the structure of the EXIGrammar. In case of simple type grammar the only thing that the EXIGrammar needs to store is the ValueType structure. In the case of complex type grammar the EXIGrammar needs to store the grammar rules. This could be solved by using a union:

```
struct EXIGrammar
{
    /**
     * Beside the other properties, this will contain whether the grammar is
     * simple or complex
     */
    unsigned char props;
    union {
        ComplexGrammar complGr; // Grammar productions and rules
        ValueType simpleGr;     //  for simple type grammars
    } grammar;
};
```

Robert[1]: Do you mean:

`ComplexGrammar complGr; // Grammar productions and rules`

or:

`ComplexGrammar* complGr; // Pointer to grammar productions and rules`

Declaring the structure inline and not through a pointer could save some space but it also means the EXIGrammar struct is always going to have maybe 4 or 8 extra bytes in to hold the two Index values, as a ValueType should pack into 4 bytes (assuming typical alignment). So it might be better to have the ComplexGrammar struct separate

Rumen[1]: I meant `ComplexGrammar complGr;` as I am thinking of having the following array in the EXIPSchema structure:

```
struct SchemaGrammarTable {
#if DYN_ARRAY_USE == ON
    DynArray dynArray;
#endif
    EXIGrammar* grammar;
    Index count;
};
```

If we have pointers in the EXIGrammar then we need to store the structures separately. Not sure what is best though. In any case having extra bytes in case we have a `ValueType simpleGr;` is still much less size than having the whole EXI grammar for the simple type.

Robert[2]: OK, understood - let's do it as you suggest

```
struct ComplexGrammar
{
    GrammarRule* rule; // Array of grammar rules which constitute that grammar
    Index count; // The size of the array
    Index contentIndex;
};
```

One problem in using a union is that you need to be able to initialize particular fields in the union which is a C99 feature that the notorious MS VS does not support. One solution to that is using void* instead of union:

```
struct EXIGrammar
{
    /**
     * Beside the other properties, this will contain whether the grammar is
     * simple or complex
     */
    unsigned char props;
```

```
    /**
     * In case of simple type grammar the void* pointer points to a ValueType
     * structure. In case of complex type grammar it points to ComplexGrammar
     */
    void* grammar;
};

struct ComplexGrammar
{
    GrammarRule* rule; // Array of grammar rules which constitute that grammar
    Index count; // The size of the array
    Index contentIndex;

};
```

Robert[1]: MS VS does allow union initialization but it has to be the first declared member.

So there is another workaround:

```
struct EXIGrammarComplexGr
{
    unsigned char props;
    union {
        ComplexGrammar complGr; // Grammar productions and rules
        ValueType simpleGr;     // <THE ACTUAL TYPE> for simple type grammars
    } grammar;
};

struct EXIGrammarSimpleGr
{
    unsigned char props;
    union {
        ValueType simpleGr;     // <THE ACTUAL TYPE> for simple type grammars
        ComplexGrammar complGr; // Grammar productions and rules
    } grammar;
};
```

The two structs are effectively the same and occupy the same storage, so you just choose the appropriate one and initialize accordingly.

So if you wanted to initialize a complex grammar:

```
struct EXIGrammarComplexGr myEXIGrammarComplexGr = { 0xAA, { &myGrammarRule, 2, 4 } };
```

and if you wanted to initialize a simple value type:

```
struct EXIGrammarSimpleGr myEXIGrammarSimpleGr = { 0xAA, { 7, 36 } };
```

If you want to declare a generic type, pick either:

```
typedef struct EXIGrammarSimpleGr EXIGrammar;
```

Rumen[1]: I like that - looks very intuitive actually.

NOTE: In case of STRICT FALSE mode, there might be a cases when a simple type grammar is pushed on the grammar processing stack and then after a AT (*) or SE (*) events another simple type grammar needs to be pushed on the stack. However, if we use just one instance of the simple type grammar, the second change of <THE ACTUAL TYPE> will erase the first one and this will create a very hard to detect bug. If this solution is implemented, during the EXI processing there should be a stack which stores the currently used <THE ACTUAL TYPE> and this stack will follow the expansion and shrinkage of the grammar stack.

Problem 4: Extending/restricting from user derived simple types

Description

This problem occurs when building the grammars in *treeTableToGrammars.c* When building a grammar for types that extend/restrict from user derived simple types you need to be able to locate the restrictions of the supertype.

This currently is not possible as there is no connection between the QNameID of the type and its restrictions stored in `SimpleTypeTable simpleTypeTable;`

Suggestion

The suggested solution of Problem 3 should solve that issue as well

NOTE:

Problem 5: Structure "Production" can be smaller

DEPRECATED! The solution of the scope issue overrides this suggestion. The only optimization that can be done is to move the EXI data type indicator (EXIType exiType;) to the SimpleTypeTable entries.

Description

Currently the grammar Production is defined as follow:

```
struct Production
{
    EXIEvent evnt;
    /**
     * For SE(qname), SE(uri:*), AT(qname) and AT(uri:*). Points to the qname of the element/attribute
     */
    QNameID qname;
    Index nonTermID; // unique identifier of right-hand side Non-terminal
};

typedef struct Production Production;
```

Where the EXIEvent contains the event type and the value type:

```
struct EXIEvent
{
    EventType eventType;
    ValueType valueType;
};

typedef struct EXIEvent EXIEvent;
```

where the ValueType is defined as follow:

```
struct ValueType
{
    EXIType exiType;
    Index simpleTypeId; // An index of the simple type in the schema SimpleTypeArray if any, INDEX_MAX otherwise
};
```

The value type is only needed in case of AT or CH events.

Suggestion

The suggested solution of Problem 3 will enable a lookup of the ValueType in the string tables during processing of CH events. If we include a EXIGrammar entries in the LocalEntry local scope table (Problem 1) for AT qnameids as well - then we can lookup the ValueType of AT events as well.

In this case the production will have the following structure without the ValueType:

```
struct Production
{
    EventType eventType; // (enum EVENT_SD, EVENT_SE_QNAME, EVENT_AT_ALL etc.)
    QNameID qname;
    Index nonTermID; // unique identifier of right-hand side Non-terminal
};
```

This will save `2*sizeof(Index)` for every production!

NOTE: In STRICT FALSE mode, there are both CH [typed] and CH [untyped] as well as AT [typed] and AT [untyped] in a single grammar rule. However, they still can be uniquely identified by the length of the event codes:

- AT(qname) [typed] have event codes with length 1 (part[0] of the rule)

- AT(qname) [untyped] have event codes with length 3 (part[2] of the rule)
- AT(*) [typed] have event codes with length 2 (part[1] of the rule)
- AT(*) [untyped] have event codes with length 3 (part[2] of the rule)
- CH [typed] have event codes with length 1 (part[0] of the rule)
- CH [untyped] have event codes with length 2 (part[1] of the rule)

Problem 6: The default entries for the string tables are stored multiple times

Description

When serializing a schema, the pre-populated string table entries and simple types are added to the EXIPSchema object even though they are already known i.e. memory for them is allocated in sTables.c and builtInGrammars.c

Suggestion

Have the pre-populated string table entries and simple types created once statically and used both for processing and not include them in the EXIPSchema object.

Rumen: It is not that easy. If we use static string table entries then it won't be possible to work on two streams simultaneously. Even the EXIOptions document will make some modifications to the string tables such as adding local value entries and new uri entry etc.
The same is valid for the grammars - they need to be augmented accordingly
It seems to me this problem is not worth solving.

Problem 7: All productions of the document grammars are created in RAM

Description

The document grammars (both schema-less and schema-informed) are created in the RAM after the EXI header is parsed.

Suggestion

Still some grammar productions need to be created dynamically as they depend on certain options in the EXI header. However, the DocContent can mostly be created statically. This includes all SE(qname) events for schema-informed grammars which will also remove the need for GlobalElemGrammarTable in the EXIPSchema.

This will also remove the need for GlobalElemGrammarTable in the EXIPSchema object.

Problem 8: Use of dynamic arrays for schema-informed tables

Description

`LnEntry` contains a `VxTable` entry, which produces the following entry in the grammars for every `LnEntry`:

```
{{sizeof(VxEntry), 0, 0, {NULL, 0}}, NULL, 0}
```

This is due to using dynamic arrays. This takes up a lot of unnecessary space. There is also other use of dynamic arrays in tables which is less critical from a space-saving point of view. Whilst the structures do have the `#if DYN_ARRAY_USE == ON` bracketing, this is not the easiest way to distinguish the tables used for grammar generation and those used for schema-informed grammars.

Suggestion

Use specific static tables that can be used by the serializer and parser but are not used by the grammar generator.

Rumen: What about using

```
#if DYN_ARRAY_USE == ON
```

During grammar creating you would need the dynamic arrays turned on. In the static code however, you would have something like this:

```
static CONST LnEntry ops_LnEntry_4[39] =
{
    {
        {
    #if DYN_ARRAY_USE == ON
        {sizeof(VxEntry), 0, 0, {NULL, 0}},
    #endif
         NULL, 0
        },
        {ops_LN_4_0, 9},
        -1, -1
    },
...
}
```

In this way the static code would work for both dynamic and static arrays and the bootstraping hell will be avoided to some extend.

Plan for working on these problems

Note that most of the suggested solutions are dependent on each other. On the other hand working on all of them simultaneously will be quite challenging. The best approach will be to take one step at a time while still try to avoid fixing same issues multiple times during the application of different solutions to the problems above.

I suggest that first we fix Problem 2: Problems with handling of the grammar serialization. That will make the refactoring of the Grammar structures much easier later on.

Then we work on solving the Problem 1: Scope issue. That will require extensive changes all over the code - grammar creation, serialization and the exi processing.

Then we work on Problem 3: Simple type grammars are redundant. This will make the fix to "Problem 4: Extending/restricting from user derived simple types" easier.

Then Problem 6: The default entries for the string tables are stored multiple times

Any comments and feedback are welcome!