Blink Schema Specification

beta5 - 2015-06-08

This document specifies a shema language for defining the structure of Blink data messages.

Contents

1 Overview

The structure of a Blink message is defined in a *schema*. Formally the schema is a logical concept but this specification defines and uses a concrete schema syntax. Implementations and users of the Blink protocol are encouraged, but not required, to support and use this syntax.

The following is an example of a schema defining a message type **Hello** with a single string field **Greeting**:

```
Hello -> string Greeting
```

2 Group

A Blink message comprises a group of fields. In the schema this is refered to as a *group type*. In the schema syntax, a group type is defined by the group name followed by an arrow (**->**) followed by a comma-separated list of field definitions. Each field definition has a type and a name:

```
Order ->
  string  Symbol,
  decimal Price,
  decimal Volume
```

Logically the fiels of a group are unordered. However, a particular message encoding may treat the order of the field declarations in a group as significant.

The schema syntax supports single inheritance. A group can specify a supergroup by following the group name by a colon and a reference to the supergroup:

```
Shape -> ...        # Base
Rect : Shape -> ... # Inherits all fields from Shape
```

A field can be marked optional to indicate that its value can be absent in the group. An optional field is followed by a question sign (**?**) in the schema syntax:

```
string Comment? # Optional field
```

For the full definition of the schema syntax, see Section 4 (page 4) .

3 Data Types

The following sections define the available data types.

3.1 Integer

Integers are signed or unsigned and are available in the widths 8, 16, 32 and 64 bits.

In the schema syntax, the integer types are named by the letter **u** for unsigned and **i** for signed, followed by the bit width. For example: **u8** and **i32** would mean unsigned 8-bit and signed 32-bit integers respectively.

3.2 String

A string logically represents a variable length sequence of Unicode characters.

In the schema syntax, the string type is named **`string`**.

An optional max size property can be specified on the string type to indicate the maximum number of bytes it can store. The maximum size is specified as an unsigned integer enclosed in parentheses. The size in bytes refers to the size of the string when encoded as [UTF-8].

```
string        # Plain Unicode string
string (17)   # String with max size 17
```

3.3 Binary

A binary value is a variable length array of bytes.

In the schema syntax, the binary type is named **`binary`**.

An optional max size properety can be specified on the binary type to indicate the maximum number of bytes it can store. The maximum size is specified as an unsigned integer enclosed in parentheses.

It is recommended that binary types are annotated to specify the underlying type more precisely using the **`@blink:type`** annotation. See Section 4.5 (page 7) .

3.4 Fixed

A fixed value is a fixed length array of bytes.

In the schema syntax, the fixed type is named **`fixed`** and is followed by the fixed size enclosed in parentheses.

It is recommended that fixed types are annotated to specify the underlying type more precisely using the **`@blink:type`** annotation. See Section 4.5 (page 7) .

Given this definition:

```
inetAddr = @blink:type="InetAddr" fixed (4)
```

an address can be encoded like this:

```
3e 6d 3c ea // 62.109.60.234
```

3.5 Enumeration

An enumeration type is a fixed set of symbols representing distinct signed 32-bit integer values.

In the schema syntax, an enumeration is a sequence of symbols separated by a bar (`|`) character. Enumerations may only appear on the right hand side of a type definition. This means that they cannot appear directly in a field definition.

```
Size = Small | Medium | Large
```

A symbol can have an explicit or implicit integer value. Explicit integer values are specified by appending a slash (`/`) and the value after the symbol:

```
Size  = Small/38 | Medium/40 | Large/42
Color = Red/0xff0000 | Green/0x00ff00 | Blue/0x0000ff
Month = Jan/1 | Feb | Mar ...
```

An implicit symbol value will have the value of the preceding symbol incremented by one. If there is no preceding symbol, then the implicit value is zero. A value may alternatively be specified as a hex number.

An enumeration with only a single symbol, which should be uncommon, is specified with a leading bar character. This is needed to make it distinct from the syntax of a type reference:

```
Singleton = | Lonely
```

3.6 Boolean

A Boolean value has the logical values true and false.

In the schema syntax, the Boolean type is named **`bool`**.

3.7 Decimal

A decimal number comprises a signed 64-bit integer significand *S* and a signed 8-bit integer exponent *E*. The value of a decimal field is:

$$S \cdot 10^{E}$$

In the schema syntax, the decimal type is named **`decimal`**.

3.8 Fixed Decimal

A fixed decimal comprises a signed 64-bit integer significand *S* and a fixed unsigned 8-bit integer exponent *E*. The exponent is specified as a non-negative integer on the type in the schema, that is, the number of decimal places. The actual value is obtained by the following calculation:

$$S \cdot 10^{-E}$$

In the schema syntax, the fixed decimal type is named **`fixedDec`** and is followed by the fixed number of decimals in parentheses:

```
price = fixedDec (2)
```

3.9 Floating Point

A floating point value is a double precision 64-bit floating point number as defined in IEEE 754-2008.

In the schema syntax, the floating point type is named **f64**.

3.10 Arbitrary Precision Number

An arbitrary precision number comprises an arbitrary large signed integer significand *S* and a signed 32-bit integer exponent *E*. The value of a decimal field is:

```
S · 10^E
```

In the schema syntax, this type is named **number**.

This type differs from the **decimal** type in that the exponent has a wider value range and that the significand can have arbitrary precision.

An optional precision property can be specified on the number type to indicate the maximum size of the significand. The maximum size in bytes is specified as an unsigned integer enclosed in parentheses.

```
number        # Variable precision
number (12)   # 96-bit significand
```

3.11 Time

A timestamp is a signed 64-bit integer that represents the time elapsed since the UNIX epoch: 1970-01-01 00:00:00.000000000 UTC. A negative timestamp indicates a point in time before the epoch. A timestamp uses either millisecond or nanosecond precision.

In the schema syntax, a timestamp is specified as **nanotime** or **millitime** for nanosecond and millisecond precisions respectively.

3.12 Date

A date is a signed 32-bit integer that represents the number of days since the Blink date epoch: 2000-01-01. A date is symbolic in the sense that it does not imply any specific timezone. It is up to the application to define how a particular date value is to be interpreted if used to specify a specific point in time.

Given a date, the number of days since the date epoch is calculated according to a proleptic Gregorian calendar [GREG]. Proleptic means that the calculation rules extend to dates before the Gregorian calender was defined. A method for date conversion is outlined in Appendix C (page 11) .

In the schema syntax, the date type is named **date**.

3.13 Time of Day

The time of day is represented as the number of milliseconds or nanoseconds since midnight. The value is represented as an unsigned 32-bit integer in the millisecond case, and as an unsigned 64-bit integer in the nanosecond case.

A time of day value must not exceed 24 hours.

A time of day value is symbolic in the sense that it does not imply any specific timezone. It is up to the application to define how a particular time of day value is to be interpreted if used to specify a specific point in time.

In the schema syntax, a time of day type is named **timeOfDayMilli** and **timeOfDayNano** for the millisecond and nanosecond precisions respectively.

3.14 Sequence

A sequence is a variable length array of items. An item can be of any value type, except for sequence. All sequences except sequences of dynamic groups (page 3), are homogeneous, that is, all items share the same type. In sequences of dynamic groups with a specified base type, all items share this same base type, but the actual type can vary between items. In sequences of type **object**, there are no constraints on the actual group type of an item.

In the schema syntax, a sequence type is indicated by two brackets (**[]**) following a type specifier.

```
u32 []     # Sequence of unsigned integers
string []  # Sequence of strings
Thing []   # Sequence of static Thing groups
Gadget* [] # Sequence of dynamic Gadget groups
object []  # Sequence of dynamic groups of any type
```

3.15 Static Group

A static subgroup is a direct or indirect reference to a group type and simply forms a logical grouping of subfields.

In the schema syntax, the use of a group type is indicated by specifying the name of the group or a type definition that resolves to a group type.

In this example:

```
StandardHeader ->
  u64       SeqNo,
  millitime SendingTime

MyMessage ->
  StandardHeader Header,
  string         Text
```

the **Header** field has a static group value of type **StandardHeader**.

3.16 Dynamic Group

A dynamic subgroup is a direct or indirect reference to a group type. The type of the actual value of a dynamic subgroup can be the referenced group type or any other type that directly or indirectly inherits from this type.

In the schema syntax, the use of a dynamic group is indicated by an asterisk (*****) following the name of a group type reference.

The schema syntax also has a type named **object** that specifies a value that can hold a dynamic group of any type.

An implementation must provide a way to determine the *actual type* used for a dynamic subgroup as oppsed to the *declared type* specified in the schema.

In this example:

```
Shape ->
  decimal Area

Rect : Shape ->
  u32 Width, u32 Height

Circle : Shape ->
  u32 Radius

Canvas ->
  Shape* [] Shapes
```

the **Shapes** field is a sequence of dynamic groups. Each item in the sequence can be of any type that inherits from **Shape**. It can for example be a mix of **Rect** and **Circle** items.

4 Schema Syntax

This section defines the overall schema semantics and schema specific artifacts. The schema syntax of the individual data types is defined in the corresponding subsections defining the encoding of each type in Section 3 (page 1) .

The following grammar [EBNF] summarizes the schema syntax. The full grammar is available here: Appendix A (page 7) .

```
schema   ::= nsDecl? def*
nsDecl   ::= "namespace" name
def      ::= define | groupDef
define   ::= name "=" (enum | type)
groupDef ::= name ("/" id)? (":" qName)?
             ("->" fields)?
fields   ::= field ("," field)+
field    ::= type name "?"?
type     ::= single | sequence
single   ::= ref | time | number | string | binary |
             fixed | "bool" | "object"
sequence ::= single "[" "]"
string   ::= "string" ("(" uInt ")")?
binary   ::= "binary" ("(" uInt ")")?
fixed    ::= "fixed" "(" uInt ")"
ref      ::= qName | qName "*"
number   ::= "i8" | "u8" | "i16" | "u16" | "i32" |
             "u32" | "i64" | "u64" | "f64" |
             "decimal" | bigNum | fixedDec
fixedDec ::= "fixedDec" "(" uInt ")"
bigNum   ::= "number" | "number" "(" uInt ")"
time     ::= "date" | "timeOfDayMilli" |
             "timeOfDayNano" | "millitime" |
             "nanotime"
enum     ::= "|" sym | sym ("|" sym)+
sym      ::= name ("/" val)?
val      ::= int | hexNum
id       ::= uInt | hexNum
qName    ::= name | cName
name     ::= (ncName - keyword) | "\" ncName
keyword  ::= "i8" | "u8" | "i16" | "u16" | "i32" |
             "u32" | "i64" | "u64" | "f64" |
             "decimal" | "fixedDec" | "number" |
             "timeOfDayMilli" | "timeOfDayNano" |
             "millitime" | "nanotime" | "date" |
             "bool" | "string" | "object" |
             "namespace" | "type" | "schema"
cName    ::= ncName ":" ncName
ncName   ::= [_a-zA-Z] [_a-zA-Z0-9]*
hexNum   ::= "0x" [0-9a-fA-F]+
int      ::= "-"? uInt
uInt     ::= [0-9]+
```

A schema is a sequence of group and type definitions. The order of the definitions in a schema is not significant. This means that a definition is allowed to refer to a definition appearing later in the same schema file. An application accepting schema files should extend the unordered property across multiple schemas. This means that a definition in one schema can refer to a type defined in another schema available in the same application.

Group definitions are the most central part of the schema since they ultimately define the structure of the messages in a protocol based on Blink. A group definition lists the fields and their types. A group can optionally inherit from a supergroup. Only a single supergroup can be specified.

The simplest possible group definition, albeit perhaps not the most useful one, is an empty group. An empty group has no fields and no supergroup and is simply specified as the group name:

```
MyEmptyMsg
```

The fields of a group follow after an arrow (**->**) following the group name. Each field is a pair comprising a type specifier and a field name. Fields are separated by a comma (**,**):

```
DbCommand ->
  u64    SeqNo,
  Action Action,
  u32    Id,
  string Value?
```

Fields can be mandatory or optional. A field is mandatory by default. A field followed by a question sign (**?**) is optional. Optional fields are encoded using the nullable format of the specified type.

A group can specify a single supergroup reference following a colon (**:**) following the group name:

```
Shape ->
  decimal Area

Rect : Shape ->
  u32 Width, u32 Height

Circle : Shape ->
  u32 Radius
```

The group that inherits from the supergroup will have all the fields, direct or indirect through inheritance, of its supergroup appearing before its own fields when encoded.

Types can be given names through type definitions. A type definition is a name followed by an equal sign (**=**), followed by a type specifier. The definition makes it possible to refer to the type specifier through that name.

```
Color =  Red | Green | Blue # Enum
Colors = Color []           # Sequence of colors
Price =  decimal            # Price is a decimal
```

Comments are allowed in a schema. They start with a pound sign (**#**) and extend to the end of the line:

```
# This is a comment
```

4.1 Constraints

Definitions must have unique names. Type definitions and group definitions are treated the same in this regard. This means that if two definitions share the same namespace, then they must not share the same name.

```
Color = Red | Green | Blue
Color -> # Ambiguous
  u32 Red, u32 Green, u32 Blue
```

Field names must be unique within a group and a field name must not shadow any inherited field.

```
Base ->
  string Field1
Derived : Base ->
  string Field1 // Error, shadows Base.Field1
```

A sequence type specifier must not directly, nor indirectly through a type reference, specify a sequence as the item type.

```
Matrix = u32 [] [] # Not allowed

Row =    string []
Table =  Row []    # Not allowed
```

A type reference used when specifying a supergroup or when specifying a dynamic group type must resolve to a group type.

```
Foo = u32
Bar : Foo         # Cannot inherit from an u32
Baz -> Foo* Data # Error, Foo is not a group
```

The symbols within an enumeration must have unique names and the values of the symbols must be distinct:

```
Month = Jan/1 | Feb | Mar/2 # Value 2 is ambiguous
```

A type definition must not directly or indirectly through chained references refer to itself.

A group definition must not directly or indirectly refer to itself through a supergroup or field type reference, unless at least one step in the chain of references is specified as dynamic.

A reference to a supergroup must not be dynamic, and it must not refer to a sequence.

The grammar for the lexical structure specifies that an integer or hex token can end in a non-numerical suffix: *numSuffix*. It is an error if such suffix is not empty. It is only present in the grammar to prevent names to follow directly after a number without any punctuation or whitespace in between.

4.2 Names and Namespaces

A schema may optionally start with a namespace declaration:

```
namespace Fix

Sym = string
Px  = decimal
Qty = decimal
```

All type and group definitions in a schema file that has a namespace declaration belongs to that namespace. Namespaces are not needed in references within a schema, but become important when referring to types across schemas and in other external contexts. It is recommended that all real world schemas carry a declaration with a short, descriptive, namespace.

Definitions in a schema lacking a namespace declaration belong to the *null namespace*.

A type reference can be qualified by a namespace by prefixing the name with the namespace name and a colon (**:**):

```
EnterOrder ->
  Fix:Sym Symbol,
  Fix:Px  Price,
  Fix:Qty Volume
```

A type reference with an unqualified name will first be resolved using the same namespace as the schema where it appears. If there is no match using this namespace, it will try to find a matching type in the null namespace.

Given the following two schemas:

```
Type1 = u8 # 1
Type2 = u8 # 2
Type3 = u8 # 3
```

and

```
namespace Ns1
Type3 = u32 # 4
```

references are resolved as indicated in the comments below:

```
namespace Ns1

Type1 = u32  # 5

Test ->
   Type1 f1, # uses 5 (5 shadows 1)
   Type2 f2, # uses 2
   Type3 f3  # uses 4 (4 shadows 3)
```

Names matching the non-terminal **`keyword`** in the grammar cannot be used directly as names in definitions and references. By preceding a keyword by a backslash (\), the keyword is quoted and can be used as a name.

```
decimal  -> i32 exp, i64 mant  # Not allowed
\decimal -> i32 exp, i64 mant # OK
```

4.3 Type Identification

In the schema, a group type is uniquely identified by a name and a namespace. However, in many situations, like in binary formats, it is more appropriate to have a numerical identifier. For this purpose this specification defines a default numerical type identifier in terms of the structure of a group definition: Appendix B (page 9) .

The schema syntax also allows an explicit numerical identifier to be specified directly on the group definition. If present, the identifier follows after a slash (`/`) after the name of the group:

```
DbCommand/9 ->
  u64 SeqNo,
  ...
```

The identifier may also be specified in a hexadecimal notation:

```
TypeWithHashBasedId/0xc36e5dfa9bc0af3d
```

From the perspective of this specification, such an identifier is just an annotation. It is up to a particular format to specify if an explicit numerical identifier is significant or not.

4.4 Annotations

Most components of a schema can be extended with annotations. Annotations do not affect how messages are encoded but carry additional information to be consumed by applications or humans.

An annotation starts with an at sign (`@`) followed by a name and a string value. The name may optionally be prefixed by a namespace name. The namespace names used in annotations are in no way related to the namespaces used in the schema itself.

There are two kinds of annotations: *inline* and *incremental*. Inline annotations directly precede the annotated component in the schema:

```
@doc="An annotated group definition"
Group1 ->
  string Text

Group2 ->
  @doc="An annotated type" string Text

Group3 ->
  string @doc="An annotated field" Text
```

Multiple inline annotations can precede the same component:

```
@doc="Initiates a session"
@code:class="Session::Logon"
Logon -> string User, string Password
```

If the same annotation name appears more than once in a sequence of annotations, later occurrences have higher precedence than earlier.

Long annotation value literals can be split into smaller literals. The value of the annotation is the concatenation of all the literal parts:

```
@long="The quick brown fox "
      "jumps over the lazy dog"
```

Field and type definition names can carry a special numeric identifier just in the same way as the type identifier can be specified on group definitions. However, these identifiers do not have any special meaning defined by this specification and are regarded as annotations.

```
Symbol/55 = string
Logout -> string Text/58
```

The incremental method allows annotations to be specified out-of-line. They can be specified in the same schema or in a schema different from the component they annotate. An incremental annotation starts with a *component reference* followed by a left arrow (`<-`) followed by a sequence of annotations separated by left arrows. Both name-value annotations and numerical identifiers can be specified this way.

The component reference is a type reference followed by an optional field or enum symbol name separated by a dot (`.`):

```
Msg -> string Payload
```

and

```
Msg <- 4711 <- @doc="A simple message"
Msg.Payload <- @doc="The data"
```

has the same meaning as

```
@doc="A simple message"
Msg/4711 -> string @doc="The data" Payload
```

To annotate the type specifier of a type definition or a field, the type reference or field name can be followed by the keyword **type** following a dot:

```
ShortStr = string
```

and

```
ShortStr.type <- @code:maxSize="10"
```

has the same meaning as

```
ShortStr = @code:maxSize="10" string
```

If the type reference resolves to a type definition and there is a name following the reference, then the type definition should contain an enumeration, and the name points out the symbol to which the annotation applies:

```
Color = Red | Green | Blue
```

and

```
Color.Blue <- @deprecated="yes"
```

has the same meaning as

```
Color = Red | Green | @deprecated="yes" Blue
```

A special form of incremental annotations start with the keyword **`schema`** and indicates that the annotations that follow apply to the schema as a whole:

```
schema <- @version="1.0" <- @author="George"
```

Implementations recognizing schema annotations are encouraged to convey any namespace that was declared in the schema where the annotation appears. This way schema annotations are allowed to be grouped by namespace.

In an application, incremental annotations are to be applied after all definitions in all schemas are known. If a definition has an inline annotation with the same name as an incremental annotation, then the incremental annotation takes precedence. The same holds for numerical identifiers. If an incremental annotation or numerical identifier is specified multiple times within the same schema file, later occurrences take precedence. If an application reads multiple schemas, then the inter-schema order of incremental annotations is undefined as far as this specification is concerned.

4.5 Derived Type Annotation

The annotation **`blink:type`** can be added to a type in the schema to specify a specialization. The value of the annotation is a name of the derived type.

The purpose of the annotation is to indicate additional constraints and semantics that can be utilized by a particular application.

A library of standard derived types is maintained at http://blinkprotocol.org/. If possible, it is recommended that types are selected from this library to increase interoperability.

This is an example of some of the standard derived types:

```
uuid =   @blink:type="UUID" fixed (16)
xml =    @blink:type="XML" string
```

A Schema Grammar

In this grammar the letter ***e*** means *empty*. The escape sequence **`\n`** means newline, and **`\xNN`** refers to a character code by two hex digits. In all other contexts a backslash is treated literally. Whitespace and comments are allowed between tokens. The tokens in the lexical structure part below are treated as single tokens and may not contain internal whitespaces or comments.

```
schema ::=
   defs
 | nsDecl defs

nsDecl ::=
   "namespace" name

defs ::=
   e
 | def defs

def ::=
   annots define
 | annots groupDef
 | incrAnnot

define ::=
   nameWithId "=" (enum | (annots type))

groupDef ::=
   nameWithId super body

super ::=
   e
 | ":" qName

body ::=
   e
 | "->" fields

fields ::=
   field
 | field "," fields

field ::=
   annots type annots nameWithId opt

opt ::=
   e
 | "?"

type ::=
   single | sequence

single ::=
   ref | time | number | string | binary | fixed |
   "bool" | "object"

sequence ::=
   single "[" "]"

string ::=
   "string"
 | "string" size

binary ::=
   "binary"
 | "binary" size

fixed ::=
   "fixed" size

size ::=
   "(" uInt ")"

ref ::=
   qName
 | qName "*"

number ::=
   "i8" | "u8" | "i16" | "u16" | "i32" | "u32" |
   "i64" | "u64" | "f64" | "decimal" | bigNum |
   fixedDec

fixedDec ::=
   "fixedDec" size

bigNum ::=
   "number"
 | "number" size

time ::=
   "date" | "timeOfDayMilli" | "timeOfDayNano" |
   "millitime" | "nanotime"

enum ::=
   "|" sym
 | sym "|" syms

syms ::=
   sym
 | sym "|" syms

sym ::=
   annots name val

val ::=
   e
 | "/" (int | hexNum)

annots ::=
   e
 | annot annots

annot ::=
   "@" qNameOrKeyword "=" literal

literal ::=
   literalSegment
 | literalSegment literal

nameWithId ::=
   name id

id ::=
   e
 | "/" (uInt | hexNum)

incrAnnot ::=
   compRef "<-" incrAnnotList

compRef ::=
   "schema"
 | qName
 | qName "." "type"
 | qName "." name
 | qName "." name "." "type"

incrAnnotList ::=
   incrAnnotItem
 | incrAnnotItem "<-" incrAnnotList

incrAnnotItem ::=
   int | hexNum | annot
```

A.1 Lexical Structure

When reading a schema, the sequence of characters is split into tokens by repeatedly finding the longest subsequence of characters that matches:

- any of the literal string terminals in the grammar above,
- the non-terminals *name*, *cName*, *hexNum*, *int*, *uInt*, *literalSegment*; or
- the non-terminal *separator*

The sequence of tokens is then matched against the grammar above. Tokens that match the non-terminal *separator* are ignored.

```
qName ::= name |
     cName

qNameOrKeyword ::=
   qName | keyword

name ::=
   (ncName - keyword)
 | "\" ncName

keyword ::=
   "i8" | "u8" | "i16" | "u16" | "i32" | "u32" |
   "i64" | "u64" | "f64" | "decimal" | "fixedDec" |
   "number" | "timeOfDayMilli" | "timeOfDayNano" |
   "millitime" |  "nanotime" | "date" | "bool" |
   "string" | "binary" | "fixed" | "object" |
   "namespace" | "type" | "schema"

cName ::=
   ncName ":" ncName

ncName ::=
   nameStartChar nameChars

nameChars ::=
   nameChar
 | nameChar nameChars

nameChar ::=
   nameStartChar | digit

nameStartChar ::=
   [_a-zA-Z]

hexNum ::=
   "0x" hexDigits numSuffix

hexDigits ::=
   hexDigit
 | hexDigit hexDigits

hexDigit ::=
   [0-9a-fA-F]

int ::=
   uInt
 | "-" uInt

uInt ::=
   digits numSuffix

digits ::=
   digit
 | digit digits

digit ::=
   [0-9]

numSuffix ::=
   e
 | nameChars

literalSegment ::=
   quoteStrLit | aposStrLit

quoteStrLit ::=
   '"' strNoQuote '"'

aposStrLit ::=
   "'" strNoApos "'"

strNoQuote ::=
   e
 | [^"\n] strNoQuote

strNoApos ::=
   e
 | [^'\n] strNoApos

separator ::=
   [ \n\t] | "#" restOfLine

restOfLine ::=
   e
 | [^\n] restOfLine
```

B Default Type Identifier

B.1 Overview

The numerical identifier defined by this specification is a 64-bit unsigned integer. The identifier for a group definition in the schema results from hashing the *signature string* of the definition. The signature string is based on the structure of the definition and any other types it directly or indirectly refers to.

B.2 Hash Function

The identifier is obtained by applying a hash function [SHA-1] to the signature string and extracting the 64 most significant bits.

Assuming the following schema:

```
namespace Eg

Hello ->
  string Greeting
```

then the signature and SHA-1 hash of the type **Eg:Hello** would become:

```
sha1 ("Eg:Hello>>UGreeting!") =
  0x55c2102b037b0a5e32f32709f2719460d3f5affe
```

from which we obtain the type identifier by extracting the 64 most significant bits:

```
0x55c2102b037b0a5e
```

B.3 Signature Translation

The translation from a schema definition to a signature is specified by an annotated grammar. The grammar is a subset of the full schema grammar specified in Appendix A (page 7) . This subset omits the syntax for annotations since they do not contribute to the signature.

The non-terminals *name*, *qName*, *uInt*, *sym* and *syms* are as defined in Appendix A (page 7) .

Each production in the grammar can have several alternatives. Each alternative has a string value. The string value is defined by an expression enclosed in curly braces. If the alternative consists of a single non-terminal and the expression is omitted, then the value is that of the non-terminal.

The signature of a definition in the schema is the string value that results from applying the *def* production.

A non-terminal can be labelled by a subscript variable name. This variable resolves to the string value of the corresponding non-terminal when used inside the expression.

A string is a sequence of UTF-8 encoded Unicode characters.

The string values of the non-terminals *name*, and *uInt* are the sequences of bytes that the corresponding productions matches in the input schema.

In an expression,

- ***A* + *B*** concatenates the strings *A* and *B*.
- **qName (*N*)** returns the fully qualified name of the definition named *N*. A fully qualified name comprises a namespace name followed by a colon, followed by the name of the definition. If the definition is in the *null namespace*, then no namespace prefix is used.
- **ref (*N*)** returns a 16 digit hexadecimal string representing the numerical type identifier of the definition that the name *N* resolves to. Any letters in the hexadecimal number will be in lower case.
- **refName (*N*)** returns the fully qualified name of the definition that the name *N* resolves to.

When resolving type names for the **qName**, **ref** and **refName** functions, any namespace declaration in the entity where the reference is located must be considered. This is specified in Section 4.2 (page 5)

```
def ::=
   define
 | groupDef

define ::=
   name_x "=" (enum | type)_y
    { qName (x) + "=" + y }

groupDef ::=
   name_x super_y body_z
    { qName (x) + ">" + y + ">" + z }

super ::=
   e
    { "" }
 | ":" qName_x
    { ref (x) }

body ::=
   e
    { "" }
 | "->" fields_x
```

```
     { x }

fields ::=
    field
  | field_x "," fields_y
     { x + y }

field ::=
    type_x name_y opt_z
     { x + y + z }

opt ::=
    e
     { "!" }
  | "?"
     { "?" }

type ::=
    single | sequence

single ::=
    (ref | time | number | string | binary | fixed)
  | "bool"
    { "B" }
  | "object"
    { "O" }

sequence ::=
    single_x "[" "]"
     { x + "*" }

string ::=
    "string"
     { "U" }
  | "string" size_x
     { "U" + x }

binary ::=
    "binary"
     { "V" }
  | "binary" size_x
     { "V" + x }

fixed ::=
    "fixed" size_x
     { "X" + x }

size ::=
    "(" uInt_x ")"
     { x }

ref ::=
    qName_x
     { "R" + ref (x) + ";" }
  | qName_x "*"
     { "Y" + refName (x) + ";" }

number ::=
    "i8"
     { "c" }
  | "u8"
     { "C" }
  | "i16"
     { "s" }
  | "u16"
     { "S" }
  | "i32"
     { "i" }
  | "u32"
     { "I" }
  | "i64"
     { "l" }
  | "u64"
     { "L" }
  | "f64"
     { "f" }
  | "decimal"
```

```
     { "d" }
  | bigNum
  | fixedDec

fixedDec ::=
    "fixedDec" size_x
     { "F" + x }

bigNum ::=
    "number"
     { "e" }
  | "number" size_x
     { "e" + x }

time ::=
    "date"
     { "D" }
  | "timeOfDayMilli"
     { "m" }
  | "timeOfDayNano"
     { "n" }
  | "millitime"
     { "M" }
  | "nanotime"
     { "N" }

enum ::=
    "|" sym
     { "E" }
  | sym "|" syms
     { "E" }
```

B.4 Example

```
Shape ->
  string Descr?

Rect : Shape ->
  Point UpperLeft, Point LowerRight

Circle : Shape ->
  u32 Radius

Canvas ->
  Shape* [] Shapes

Point ->
  u32 X, u32 Y
```

The following table lists the signatures and hashes for the preceding schema:

```
"Point>>IX!IY!"                                 0x00b22138bdbe9d77
"Shape>>UDescr?"                                0xb7c673c8db3f118b
"Canvas>>YShape;*Shapes!"                       0x5f1f2cdf3f11d72e
"Circle>b7c673c8db3f118b>IRadius!"              0x2a89e2228875c007
"Rect>b7c673c8db3f118b>R00b22138bdbe9d77;
UpperLeft!R00b22138bdbe9d77;LowerRight!"        0x1378e52fb385fed9
```

C Date Calculations

The following C fragment has two functions that outlines how to convert to and from a Blink date value. The **`toDays`** function converts a date expressed as a year, a month and a day of month, into the number of days from the Blink date epoch 2000-01-01. The **`toDate`** function converts in the other direction.

```c
typedef long i64;
typedef int i32;

static const i32 EpochOffset = 730425;

i32 toDays (i32 y_, i32 mm, i32 dd)
{
   i64 m = (mm + 9) % 12;
   i64 y = y_ - m / 10;
   i32 days = 365*y + y/4 - y/100 + y/400 +
      (m*306 + 5)/10 + (dd - 1);
   return days - EpochOffset;
}

void toDate (i32 days_, i32* y_, i32* mm_, i32* dd_)
{
   i64 days = days_ + EpochOffset;
   i64 y = (10000*days + 14780) / 3652425;
   i64 ddd = days - (365*y + y/4 - y/100 + y/400);
   if (ddd < 0)
   {
      -- y;
      ddd = days - (365*y + y/4 - y/100 + y/400);
   }
   i64 mi = (100*ddd + 52) / 3060;
   i64 mm = (mi + 2) % 12 + 1;
   y = y + (mi + 2) / 12;
   i64 dd = ddd - (mi*306 + 5) / 10 + 1;
   *y_ = y; *mm_ = mm; *dd_ = dd;
}
```

NOTE: The arithmetics must take place in the 64-bit space, hence the type definition for **`i64`**. In order to compile this fragment on a 32-bit architecture you would probably have to change the definition.

D References

EBNF	http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation
GREG	http://en.wikipedia.org/wiki/Proleptic_Gregorian_calendar
SHA-1	https://tools.ietf.org/html/rfc3174
UTF-8	http://tools.ietf.org/html/rfc3629