# Blink Specification

beta4 - 2013-06-14

*This document specifies a method for defining the structure of data messages and encoding these into an efficient and compact binary form.*

## Contents

## 1 Overview

The Blink compact binary encoding is a format for encoding structural data messages. The format is designed to be

- *simple* - it tries to minimize the number of encoding artifacts, primitive constructs, and the ways they can be combined,
- *efficient* - encoders and decoders can be efficiently implemented in software or hardware,
- and *compact* - encoded messages are reasonably compact due to the extensive use of variable-length coded data fields and the lack of inline meta data.

A Blink message comprises a size, a type identifier and a sequence of ordered and typed fields. Field types include integer, string, binary, Boolean, decimal, floating point, time, date, sequence and subgroup.

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 encoding of integers and types derived from integers use a Variable-Length Code (VLC) approach. The VLC encoding results in fields of variable size: numbers with smaller magnitude consume fewer bytes than numbers with greater magnitude. This gives greater flexibility to the selection of the value ranges while maintaining messages reasonably compact.

Blink allows fields to be marked as optional in the schema. Optional fields use a *nullable* encoding that is specific to the type of the field.

Blink messages are extensible and can carry unsolicited content in a controlled way. This means that decoders that do not know or do not care about an extension can ignore it. The extensions are encoded in Blink themselves.

The following is a small teaser just to show what an encoded blink message can look like. Given a schema

```
Hello/1 -> string Greeting
```

a **Hello** message carrying the greeting "Hello World" would be encoded as

```
0d 01 0b 48 65 6c 6c 6f 20 57 6f 72 6c 64
```

or annotated:

```
0d                // Msg size: 13 bytes follow
01                // Msg type Hello has ID 1
0b                // String length: 11 bytes follow
48 65 6c 6c 6f 20 // "Hello World"
57 6f 72 6c 64    // ...
```

## 2 Notation and Conventions

Encoded bytes are written as two digit hex numbers, and if they appear in paragraph text, they are prefixed by **`0x`**.

This specification specifies constraints that must or can be checked by a decoder. It specifies constraints that **must** be checked as *strong* errors and constraints that **can** be checked as *weak* errors. See Section 6 (page 6) for details.

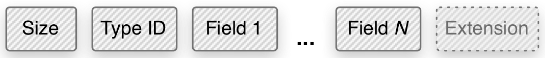
Type names written in a monospaced font like this: **`u64`**, refer to the corresponding value types in the schema language. The encoding of such values are as specified in this specification.

## 3 Message Structure

A Blink message, or more general a *group*, consists of a preamble followed by a sequence of typed *data fields*. The preamble comprises a group size followed by a numerical *type identifier*.

> **NOTE**: The terms *message* and *group* are used interchangeably. A group is the general construct and a message is simply a group used in a message passing context.

Figure 3-1 *Message Layout*



The size preamble specifies the number of bytes that follow after it. The size preamble is encoded as a **`u32`**. However, a particular application may put additional constraints on the maximum allowed size. It is a weak error (W1) if the size is zero.

The type identifier links a group instance with the corresponding definition in the schema. The type identifier is encoded as an **`u64`**. Typically you want to keep identifiers as short as possible, but the wider range is provided to allow for hash based identification schemes for example. It is a weak error (W2) if the type identifier is not known to a decoder.

The data fields are not self-describing and their order is therefore significant and must follow the order specified in the corresponding group definition in the schema.

It is a strong error (S1) if a group ends prematurely. A group ends prematurely if all bytes, as specified by the group size, have been consumed and there is still at least one non-nullable field left that has not been processed. This means that a decoder should treat a group as if it is logically extended with an infinite sequence of null values.

In the schema syntax, a group 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/17 ->
  string  Symbol,
  decimal Price,
  decimal Volume
```

In this example the group name is annotated with the type identifier 17.

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
```

For the full definition of the schema syntax, see Section 7 (page 6) .

The following sections define the available data types. In addition to the normal encoding, each data type has a nullable representation that is used when a field is specified as optional in the schema. Optional fields are followed by a question sign (**`?`**) in the schema syntax:

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

### 3.1 Integer

Integers are signed or unsigned and are available in the widths 8, 16, 32 and 64 bits. All integer values are VLC encoded, see Section 4 (page 5). Signed integers use a two's complement representation where the most significant data bit is the sign bit [TWOC].

```
40             // 64 unsigned
80 01          // 64 signed
a7 49          // 4711
c4 ff ff ff ff // 4294967295 (u32 max)
40             // -64 signed
99 b6          // -4711 signed
c4 00 00 00 80 // -2147483648 (i32 min)
```

The encoding does not depend on the specified bit width, but its serves as a dimension and range specifier for the application. However, it does matter if the type is signed or unsigned since the first bit becomes the sign bit in the former case. This is demonstrated in the example above where the integer 64 has different encodings in the signed and unsigned cases. Since the highest data bit is set in **`0x40`**, this would be interpreted as the sign bit if the value was treated as signed integer. Therefore, an extra byte is needed to make room for a zero sign bit: **`0x80 0x01`**.

It is a weak error (W3) if the decoded value overflows the range implied by the specified width. It is also a weak error (W4) if the VLC entity contains more bytes than needed to express the full width of the type, that is, if the byte count is greater than **`(width/8)+1`**.

A special VLC entity, **`0xc0`**, is used to represent NULL. The VLC meaning of this entity is an entity with no data bits and can therefore never appear when encoding a normal integer. This means that integers need no special nullable encoding.

It is a weak error (W5) if the special NULL value appears in a field that is not declared optional in the schema.

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 value is encoded as a size preamble followed by the specified number of UTF-8 encoded bytes. The size preamble is encoded as an **u32**. It is a weak error (W6) if the bytes do not form a valid [UTF-8] sequence.

A nullable string field uses a nullable size preamble.

```
05 48 65 6c 6c 6f       // "Hello"
0d 52 c3 a4 6b 73       // "Räksmörgås"
6d c3 b6 72 67 c3 a5 73 // ...
00                      // Empty string
```

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. It is a weak error (W7) if a string exceeds the specified maximum size.

The maximum size serves as a hint to implementations and can also be used by alternative encodings to indicate that a more efficient represention of the string should be used. It is however not utilized in the compact binary format defined by this specification.

```
string        # Plain UTF-8 string
string (17)   # String with max size 17
```

## 3.3 Binary

A binary value is encoded as a size preamble followed by the specified number of bytes. The size preamble is encoded as an **u32**.

A nullable binary field uses a nullable size preamble.

```
04 de ad be ef // "\xde\xad\xbe\xef"
00             // Empty binary
```

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 a weak error (W8) if a binary value exceeds the specified maximum size.

The maximum size serves as a hint to implementations and can also be used by alternative encodings to indicate that a more efficient represention of the binary should be used. It is however not utilized in the compact binary format defined by this specification.

It is recommended that binary types are annotated to specify the underlying type more precisely using the **@blink:type** annotation. See Section 7.4 (page 9) .

Given this definition:

```
bigInt = @blink:type="BigInt" binary
```

then the 13th Fibonacci prime **1066340417491710595814572169** can be encoded like this:

```
0c 03 72 0e 5d dc d8 a3 1e 44 36 c0 89
```

## 3.4 Fixed

A fixed value is encoded as a fixed-length sequence of bytes. The number of bytes is specified on the type in the schema.

A nullable fixed field is preceded by a presence flag byte. If the value is present, then the flag byte is **0x01**. It is a weak error (W9) if the flag is not **0xc0** (NULL) or **0x01**. If the presence flag is null, then it is not followed by the fixed 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 7.4 (page 9) .

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. The value is encoded as an **i32**. It is a weak error (W10) if the value does not correspond to any symbol in the schema.

A nullable enumeration field is encoded using a nullable **i32** integer.

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. A Boolean value is encoded an **u8** with the possible values 0 and 1. It is a weak error (W11) if the decoded value is not 0 or 1.

A nullable Boolean field is encoded as a nullable **u8**.

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

## 3.7 Decimal

A decimal number is encoded as a composite entity comprising a signed 8-bit integer exponent *E* followed by a signed 64-bit integer mantissa *M*. The decoded value is obtained by the following calculation:

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

A nullable decimal field is encoded using a nullable exponent. If the exponent is null, then no mantissa follows.

```
7e c2 10 27 // 10000 · 10^-2 = 100.00
```

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

## 3.8 Floating point

A floating point value is encoded as a double precision 64-bit floating point number as defined in IEEE 754-2008. The bits are encoded as an **u64**.

A nullable floating point field is encoded using a nullable **u64** integer.

```
c8 1b de 83 42 ca c0 f3 3f // 1.23456789
c8 00 00 00 00 00 00 f0 7f // Infinity
```

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

## 3.9 Time

A timestamp is encoded as an **i64**. The integer 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.

A nullable timestamp field is encoded using a nullable **i64**.

```
// 2012-10-30 00:00:00 GMT+1

c8 00 60 9c f5 04 ad c1 12 // nanotime
c6 80 c5 c0 ae 3a 01       // millitime
```

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

## 3.10 Date

A date is encoded as an **i32** representing 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 13) .

A nullable date field is encoded using a nullable **i32**.

```
8e 49 // 2012-10-30
```

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

## 3.11 Time of Day

The time of day is represented as the number of milliseconds or nanoseconds since midnight. The value is encoded as an **u32** in the millisecond case, and as an **u64** in the nanosecond case.

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.

It is a weak error (W12) if the decoded value represents 24 hours or more, that is, if it is greater than 86399999 or 86399999999999 depending on the precision.

A nullable time of day field is encoded using a nullable **u32** or **u64** depending on the precision.

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

## 3.12 Sequence

A sequence of items is encoded with an **u32** length preamble specifying the number of items that follow. An item can be of any value type, except for sequence. All sequences except sequences of dynamic groups (page 5), 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.

A nullable sequence field is encoded using a nullable length preamble.

```
03 01 02 03                // [1, 2, 3]
02 03 66 6f 6f 03 62 61 72 // ["foo", "bar"]
```

```
00                          // Empty sequence
```

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
```

## 3.13 Static Group

A static subgroup is just a logical grouping of subfields and has no additional extent of its own when encoded.

A nullable static subgroup field is preceded by a presence flag byte. If the subgroup is present, then the flag byte is **`0x01`**. It is a weak error (W13) if the flag is not **`0xc0`** (NULL) or **`0x01`**. If the presence flag is null, then the fields of the subgroup are not encoded.

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

Given these definitions:

```
StandardHeader ->
  u64       SeqNo,
  millitime SendingTime

MyMessage/2 ->
  StandardHeader Header,
  string         Text
```

and a message:

```
MyMessage:
  SeqNo:       1
  SendingTime: 2012-10-30 00:00:00 GMT+1
  Text:        Hello
```

it would be encoded as:

```
Sz Id SeqNo SendingTime          Text
0e 02 01    c6 80 c5 c0 ae 3a 01 05 48 65 6c 6c 6f
```

## 3.14 Dynamic Group

A dynamic subgroup has a size and type identifier preamble preceding the subfields. The type identifier specifies the *actual type* of the group. A dynamic group has the exact same structure as a top level message, including the ability to carry an unsolicited extension, see Section 3 (page 2). It is a weak error (W14) if the type identifier is not known to a decoder.

A nullable dynamic subgroup field is encoded using a nullable size preamble. If the length preamble is null, then no type identifier or fields follow.

In the schema syntax, the use of a dynamic group is indicated by an asterisk (`*`) following the name of a group type reference. It is a weak error (W15) if the actual type specified by the identifier appearing in the dynamic group does not refer to a type that is structurally compatible with the type specified in the schema, the *declared type*. When using the schema model defined in this specification this means that the actual type must be the same as the declared type or that the declared type is an ancestor through inheritance of the actual type.

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

Given these definitions:

```
Shape ->
  decimal Area

Rect/3 : Shape ->
  u32 Width, u32 Height

Circle/4 : Shape ->
  u32 Radius

Canvas/5 ->
  Shape* [] Shapes
```

and a message:

```
Canvas:
  Shapes: [
    Rect: Area: 6.0 Width: 2 Height: 3
    Circle: Area: 28.3 Radius: 3
  ]
```

it would be encoded as:

```
0e        // Msg size: 14
05        // Msg type Canvas has ID 5
02        // Canvas.Shapes has two items
05        // First item group has size 5
03        // Group type Rect has ID 3
7f 3c     // Rect.Area = 60 · 10^-1
02        // Rect.Width = 2
03        // Rect.Height = 3
05        // Second item group has size 5
04        // Group type Circle has ID 4
7f 9b 04  // Circle.Area = 283 · 10^-1
03        // Circle.Radius = 3
```

# 4 Variable-Length Code (VLC)

Any integer value or value that is derived from an integer, like a timestamp, is encoded with a variable-length code. Also, integers that are artifacts of the Blink encoding itself, like size preambles and type identifiers, use VLC encoding.

The VLC encoding used here is a hybrid of a prefix code at the bit level combined with a prefix byte at the byte level. This hybrid is designed to strike a balance between compactness of small integers and processing speed of larger integers.

There are three basic forms of VLC entities. Each form is identified by the leading bits of the first byte:

- The one bit prefix **`0`** indicates that the entity comprises a single byte with 7 data bits.

- The two bit prefix **10** indicates that the entity comprises two bytes and has 14 data bits where the 6 least significant bits are stored after the bit prefix in the first byte.
- The two bit prefix **11** indicates that the following six bits is an integer that specifies how many bytes with data bits that follow.

In the two multibyte forms, the entity uses little-endian byte order. That is, bytes with lower order bits precede bytes with higher order bits.

The three forms can be illustrated at the bit level like this:

**0** $b_6b_5b_4b_3b_2b_1b_0$ 7 data bits
**1 0** $b_5b_4b_3b_2b_1b_0$ $b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6$ 14 data bits
**1 1** $n_5n_4n_3n_2n_1n_0$ [$n$ bytes] $n \cdot 8$ data bits

Functions for encoding and decoding VLC entities are outlined in Appendix D (page 13) .

## 4.1 NULL

A single null value is used for all data types to indicate the absence of a field value: **0xc0**.

## 5 Message Extensions

Any message or dynamic subgroup can carry extra extension data. A decoder detects the presence of an extension when the specified size of a group is larger than the bytes consumed when all defined fields have been processed. An extension is encoded just as a sequence of dynamic subgroups. That means that the extension appears as if a field like this ended the group:

```
object [] Extension
```

A decoder can choose to ignore an extension altogether, or it can decode it and process some or all of the contained groups based on their type identifiers.

Given these definitions:

```
Mail/7 ->
  string Subject, string To, string From, string Body
Trace/8 ->
  string Hop
```

and a message:

```
Mail:
  Subject: Hello
  To: you
  From: me
  Body: How are you?
```

and two extension groups:

```
Trace:
  Hop: local.eg.org
Trace:
  Hop: mail.eg.org
```

it would be encoded as:

```
39                   // Msg size: 57 incl. extension
07                   // Msg type Mail has ID 7
05 48 65 6c 6c 6f    // Mail.Subject
03 79 6f 75          // Mail.To
02 6d 65             // Mail.From
0c 48 6f 77 20 61 72 // Mail.Body
65 20 79 6f 75 3f    // ...

// Extension:

02                   // Extension group count
0e                   // Size of first extension
08                   // Group type Trace has ID 8
0c 6c 6f 63 61 6c 2e // Trace.Hop
65 67 2e 6f 72 67    // ...
0d                   // Size of second extension
08                   // Group type Trace has ID 8
0b 6d 61 69 6c 2e 65 // Trace.Hop
67 2e 6f 72 67       // ...
```

## 6 Error Handling

There are two kinds of errors:

- *strong* - a decoder must check for strong errors. When a strong error occurs, the decoder must skip the current message being decoded. A session oriented application can also choose to terminate the session where the strong error occurs.
- *weak* - a decoder can choose to ignore a weak error and recover from it in an implementation dependent way. If a weak error is checked for and detected, it should be treated in the same way as a strong error.

An encoder must not make any assumptions about how a decoder will handle weak constraints and must comply with both strong and weak constraints.

## 7 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 2) .

The following grammar [EBNF] summarizes the schema syntax. The full grammar is available here: Appendix A (page 10) .

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

In order to encode a message or dynamic group, it must have a numerical type identifier. An implementation can have several methods of assigning identifiers to group types. The schema syntax provides one such method: a type identifier can follow a slash (`/`) following the name of a group:

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

A type identifier may also be specified in a hexadecimal notation:

```
TypeWithHashBasedId/0xc36e5dfa9bc0af3d
```

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

```
# This is a comment
```

## 7.1 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.

## 7.2 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
```

## 7.3 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.

## 7.4 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)
bigInt = @blink:type="BigInt" binary
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 "=" annots (enum | 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"

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

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" | "date" |
   "timeOfDayMilli" | "timeOfDayNano" | "nanotime" |
   "millitime" | "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 Encoding Grammar

This grammar gives an overview of the encoding structure.

In this grammar the letter ***e*** means *empty*.

```
stream ::=
    e
  | dynGroup stream

dynGroup ::=
    length typeId group extension

group ::=
    e
  | field group

nullableGroup ::=
    null
  | "\x01" group

extension ::=
    e
  | length dynGroups

dynGroups ::=
    e
  | dynGroup dynGroups

field ::=
    value | sequence | nullableGroup | nullableFixed |
    null

sequence ::=
    length items

items ::=
    e
  | value items

value ::=
    u8 | i8 | u16 | i16 | u32 | i32 | u64 | i64 |
    f64 | decimal | millitime | nanotime | date |
    timeOfDayMilli | timeOfDayNano | group | dynGroup |
    string | binary | fixed | bool

u8 ::=   vlcEntity
i8 ::=   vlcEntity
u16 ::=  vlcEntity
i16 ::=  vlcEntity
u32 ::=  vlcEntity
i32 ::=  vlcEntity
u64 ::=  vlcEntity
i64 ::=  vlcEntity
bool ::= vlcEntity
f64 ::=  vlcEntity

decimal ::=
    exponent mantissa

mantissa ::= i64
exponent ::= i8

string ::=
    length bytes

binary ::=
    length bytes

length ::= u32

fixed ::=
    bytes

nullableFixed ::=
    null
  | "\x01" fixed

millitime ::=      i64
nanotime ::=       i64
date ::=           i32
timeOfDayMilli ::= u32
timeOfDayNano ::=  u64

typeId ::= u64

bytes ::=
    e
  | byte bytes

byte ::=
    [\x00-\xff]

vlcEntity ::=
  vlcOne | vlcTwo | vlcN;

vlcOne ::=
  cc ["\x00", "-", "\x7f"];

vlcTwo ::=
  cc ["\x80", "-", "\xbf"] byte;

vlcN ::=
  cc ["\xc1", "-", "\xff"] bytes;

null ::=
  "\xc0";
```

## 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.

```
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 Integer Encoding and Decoding Functions

The following C fragment has two functions that outlines how to convert integer values to and from the VLC encoding used in the Blink compact binary format. The functions **encode_u64** and **decode_u64** translates a 64-bit integer to and from VLC.

```
typedef unsigned char u8;
typedef unsigned long u64;
typedef int  i32;
typedef long i64;

i32 get_data_size (u64 data)
{
   u64 mask = (~0ULL) >> 8;
   int p1;

   for (p1 = 8 ; p1 > 1 ; -- p1, mask >>= 8)
      if (data > mask)
         break;

   return p1;
}

i32 encode_u64 (u8* buf, u64 data)
{
   if (data < 0x4000)
   {
      if (data < 0x80)
      {
         buf [0] = data;
         return 1;
      }

      buf [0] = 0x80 | (data & 0x3f);
      buf [1] = data >> 6;
      return 2;
   }
   else
   {
      i32 size = get_data_size (data);
      buf [0] = 0xc0 | size;

      i32 p1;
      for (p1 = 1 ; p1 <= size ; ++ p1)
      {
         buf [p1] = data & 0xff;
         data >>= 8;
      }
      return size + 1;
   }
}

i32 decode_u64 (u8* buf, u64* data)
{
   if (buf [0] < 0xc0)
   {
      if (buf [0] < 0x80)
      {
         *data = buf [0];
         return 1;
      }

      *data = (buf [0] & 0x3f) | (buf [1] << 6);
      return 2;
   }
   else
   {
      i32 size = buf [0] & 0x3f;
      u64 temp = 0;
      i32 p1;

      for (p1 = 0 ; p1 < size ; ++ p1)
         temp |= (u64) buf [p1 + 1] << (8 * p1);

      *data = temp;
      return size + 1;
   }
}
```

## E References

| | |
|---|---|
| **EBNF** | http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation |
| **GREG** | http://en.wikipedia.org/wiki/Proleptic_Gregorian_calendar |
| **TWOC** | http://en.wikipedia.org/wiki/Two's_complement |
| **UTF-8** | http://tools.ietf.org/html/rfc3629 |