Blink Native Binary Format Specification

beta4 - 2013-06-05

This document specifies a method for encoding messages structured by a Blink schema into a native binary format. Encoded fields have fixed widths and appear at predictable offsets given the start of a message.

Contents

Appendices

1 Overview

The core Blink specification [BLINK] defines a schema language for specifying the structure of data messages. It also defines a compact binary format for messages defined by such schema.

This specification provides a complementary binary format called Blink Native. In this format, fields have a fixed width and a predictable offset relative to the start of the message or group.

1.1 Rationale

The native format results in less compact messages compared to the compact binary format in the core Blink specification. The advantage is that the encoding and decoding overhead is lower. This means that this format is suitable in situations where bandwidth or storage size is not a limiting factor and encoding and decoding speed is of essence.

The native format was designed to meet the following goals:

- Full fidelity to the core specification: any message encoded in the core binary format can also be represented by a message in the native format as long as the contained types are defined by a schema.
- Zero overhead to produce and access static content in environments that allow efficient access to misaligned data, also in messages that have dynamic content. This is why the format is called native.
- Minimal overhead to produce and access dynamic content.
- Make the producer of a message spend the effort of calculating the offsets to dynamic content. Allow a producer that has prior knowledge about the dynamic content to pre-calculate the offsets.

1.2 Encoding Summary

The following list summarizes the encoding rules:

- A message or dynamic group starts with a 32-bit length preamble, followed by a 64-bit type identifier, followed by a 32-bit extension offset, followed by the fields of the group.
- Integer values and values derived from integers are represented by a fixed number of bytes in little-endian order.
- Values of variable size are allocated in a data area after the fixed width fields of the group and the inline value becomes a relative offset to the actual value.
- String values can be stored inline if annotated by a maximum size in the schema.
- Optional fields are preceded by a one-byte presence flag.

1.3 Hello World

The following shows what the first example in the core Blink specification would look like in the native format:

Assuming the schema:

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

and a message:

```
@Hello|Greeting=Hello World
```

then it would be encoded as

```
1f 00 00 00       // Msg size: 31 bytes follow
01 00 00 00 00 00 00 00 // Type ID of Hello: 1
00 00 00 00       // No extension
04 00 00 00       // Greeting: relative offset: 4
0b 00 00 00       // String length: 11 bytes follow
48 65 6c 6c 6f 20 // "Hello "
57 6f 72 6c 64    // "World"
```

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 5 (page 5) for details.

Type names written in a monospaced font like this: **`u64`**, refers to the corresponding value type in the schema language specified in the core Blink specification [BLINK]. The encoding of such a value is as specified in this specification.

3 Message Structure

In the native format, a message, or more general a *dynamic group*, comprises a *size preamble*, followed by a numerical *type identifier*, followed by an *extension offset*, followed by a sequence of typed *data fields*, followed by a *data area* for variable sized values.

Figure 3-1 *Message Layout*

The *size preamble* specifies the number of bytes that follow after it. The size preamble is encoded as an **`u32`**. An application may put additional constraints on the maximum allowed size. It is a weak error (W1) if the size is less than 12 which is the combined size of the type identifier and the extension offset fields.

The *type identifier* links a group instance with the corresponding definition in the schema. The type identifier is encoded as an **`u64`**. It is a weak error (W2) if the type identifier is not known to a decoder.

The *extension offset* is encoded as an **`u32`** and is a relative offset to the actual extension in the data area. A zero offset indicates that the message is not extended. See Section 4 (page 5). It is a weak error (W3) if the location that results from resolving a non-zero extension offset is outside the data area of the group.

The following example shows what a message preamble looks like:

```
78 00 00 00             // Msg size: 120
11 00 00 00 00 00 00 00 // Type ID: 17
00 00 00 00             // No extension
...
```

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.

A field that is specified as optional in the schema is preceded by a **`bool`** value that indicates if the following field has a value or not. If the presence flag is false, then the bytes of the field must be all zero. It is a weak error (W4) if an unused byte is not zero.

Assuming a schema:

```
Bill/2 -> u32 Amount, u32 Tip?
```

and two messages:

```
@Bill|Amount=100
@Bill|Amount=1000|Tip=100
```

then it would be encoded like this:

```
15 00 00 00             // Msg size: 21
02 00 00 00 00 00 00 00 // Type ID for Bill: 2
00 00 00 00             // No extension
64 00 00 00             // Amount: 100
00 00 00 00 00          // No Tip

15 00 00 00             // Msg size: 21
02 00 00 00 00 00 00 00 // Type ID for Bill: 2
00 00 00 00             // No extension
e8 03 00 00             // Amount: 1000
01 64 00 00 00          // Tip: 100
```

In the second message, where the **`Tip`** field is present, the presence flag is set to true and the value is populated accordingly.

Values having variable size cannot be placed inline since the fields of a group must have fixed sizes. Instead, they are placed in a *data area* that follows after the last field of the group. A relative **`u32`** offset is stored in the field, referring to the place in the data area where the actual value is located. The offset is relative to the position of the first byte of the offset value itself. It is a weak error (W5) if the location that results from resolving an offset is outside the data area of the group.

Assuming a schema with a message with two variable size strings:

```
Person/3 -> string FirstName, string LastName
```

and a message:

```
@Person|FirstName=George|LastName=Blink
```

then it would be encoded like this:

```
    27 00 00 00             // Msg size: 39
    03 00 00 00 00 00 00 00 // Type ID for Person: 3
    00 00 00 00             // No extension
    08 00 00 00             // FirstName: off: 8 (1)
```

```
     0e 00 00 00             // LastName: off: 14 (2)
                             // -- data area --
(1)  06 00 00 00             // FirstName: size: 6
     47 65 6f 72 67 65       // "George"
(2)  05 00 00 00             // LastName: size: 5
     42 6c 69 6e 6b          // "Blink"
```

The order of the values in the data area is not significant. The data area may also contain bytes that are not part of any value. Such pad bytes must always be zero. It is a weak error (W6) if a pad byte is not zero.

It is a strong error (S1) if a group ends prematurely. A group ends prematurely if the actual group size is less than the combined calculated size of all its fields. The actual group size is the specified size minus 12, which is the combined size of the extension offset and the type identifier.

> **NOTE**: A decoder does not have to check the S1 constraint for subgroups until it actually uses them.

The following sections define the encoding of the available data types.

3.1 Integer

The following list shows the value sizes for the integer types available in a Blink schema:

- **u8**, **i8** - one byte
- **u16**, **i16** - two bytes
- **u32**, **i32** - four bytes
- **u64**, **i64** - eight bytes

Multibyte integers are encoded in little-endian byte order. That is, bytes with lower order bits precede bytes with higher order bits.

Signed integers use a two's complement representation where the most significant bit is the sign bit [TWOC].

```
11          // u8, val: 17
ff ff       // i16, val: -1
11 00 00 00 // u32, val: 17
```

3.2 String

A string can be represented in one of two ways: *indirectly* with variable capacity, or *inline* with fixed capacity.

An indirect string is encoded by placing the value in the data area and a relative offset to its location in the field or sequence item. The value in the data area comprises an **u32** size preamble followed by the specified number of data bytes.

The example from Section 1 (page 1) shows how an indirect string is encoded:

```
     1f 00 00 00             // Msg size: 31
     01 00 00 00 00 00 00 00 // Type ID for Hello: 1
     00 00 00 00             // No extension
     04 00 00 00             // Greeting: offset: 4 (*)
                             // -- data area --
(*)  0b 00 00 00             // Greeting: size: 11
     48 65 6c 6c 6f 20 57 6f // "Hello Wo"
     72 6c 64                // "rld"
```

A string is encoded inline if it has a known fixed capacity. An inline string comprises an **u8** size preamble followed by the number of bytes specified by the capacity. The size preamble specifies how many of the bytes that are used as actual data bytes. It is a weak error (W7) if the value of the size preamble exceeds the specified fixed capacity. Bytes that are not used for data must be set to zero. It is a weak error (W8) if an unused byte is not zero.

A string has a fixed capacity if its type specifier has a max size property and its value is between 1 and 255, inclusive.

Assuming a schema with a max size property:

```
Hello/1 -> string (12) Greeting
```

then the message from the previous example would be encoded like this:

```
19 00 00 00             // Msg size: 25
01 00 00 00 00 00 00 00 // Type ID for Hello: 1
00 00 00 00             // No extension
0b                      // Greeting: size: 11
48 65 6c 6c 6f 20 57 6f // "Hello Wo"
72 6c 64 00             // "rld" + one pad byte
```

The data bytes of a string must be a valid UTF-8 sequence. It is a weak error (W9) if the bytes do not form a valid [UTF-8] sequence.

3.3 Binary

A binary value is encoded in the same way as a string. The only difference is that there is no restriction on the encoding of the data bytes. This means that the W9 constraint does not apply to binary values.

3.4 Fixed

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

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

3.6 Boolean

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

3.7 Decimal

A decimal number is encoded as a composite entity comprising an **i8** exponent *E* and an **i64** mantissa *M*. The decoded value is obtained by the following calculation:

$$M \cdot 10^E$$

3.8 Floating point

A floating point value, **f64**, is encoded as a double precision 64-bit floating point number as defined in IEEE 754-2008. The eight bytes are placed in little-endian byte order.

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

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 as specified by the schema.

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.

See the core Blink specification [BLINK] for a description of how the date value is interpreted.

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.

3.12 Sequence

A sequence is represented as a relative offset to a location in the data area where the actual value is stored. The sequence value in the data area comprises an **u32** length preamble specifying the number of items that follow. An item can be of any value type, except for sequence. All items must share the same type. All dynamic group types are treated as the same type in this regard.

It is a weak error (W13) if the actual data area is too small to fit the number of items specified.

Assuming a schema:

```
Chart/4 -> u32 [] Xvals, u32 [] Yvals
```

and a message:

```
@Chart|Xvals=[0;10;20]|Yvals=[1;17;0]
```

then it would be encoded like this:

```
     34 00 00 00             // Msg size: 52
     04 00 00 00 00 00 00 00 // Type ID for Chart: 4
     00 00 00 00             // No extension
     08 00 00 00             // Xvals: offset: 8 (1)
     14 00 00 00             // Yvals: offset: 20 (2)
                             // -- data area --
(1)  03 00 00 00             // Xvals: size: 3
     00 00 00 00             // Xvals [0] = 0
     0a 00 00 00             // Xvals [1] = 10
     14 00 00 00             // Xvals [2] = 20
(2)  03 00 00 00             // Yvals: size: 3
     01 00 00 00             // Yvals [0] = 1
     11 00 00 00             // Yvals [1] = 17
     00 00 00 00             // Yvals [2] = 0
```

A sequence of variable sized items, like dynamic groups or indirect strings, will be a sequence of relative offsets to locations in the data area. Each offset is relative to the first byte of the offset value itself.

3.13 Static Group

A static subgroup value is a logical grouping of subfields and has no additional extent of its own when encoded. It has fixed size and is always encoded inline. It has no data area of its own but any offset fields refer to the data area of the nearest enclosing dynamic group or message.

Assuming a schema:

```
Point  -> u32 X, u32 Y
Rect/5 -> Point Pos, u32 Width, u32 Height
Path/6 -> Point [] Points
```

and two messages:

```
@Rect|Pos={X=3|Y=4}|Width=10|Height=10
@Path|Points=[X=1|Y=1;X=10|Y=2]
```

then they would be encoded like this:

```
     1c 00 00 00             // Msg size: 28
```

```
     05 00 00 00 00 00 00 00 // Type ID for Rect: 5
     00 00 00 00             // No extension
     03 00 00 00             // Pos.X: 3
     04 00 00 00             // Pos.Y: 4
     0a 00 00 00             // Width: 10
     0a 00 00 00             // Height: 10

     24 00 00 00             // Msg size: 36
     06 00 00 00 00 00 00 00 // Type ID for Path: 6
     00 00 00 00             // No extension
     04 00 00 00             // Points: offset: 4 (*)
                             // -- data area --
(*)  02 00 00 00             // Points: size: 2
     01 00 00 00             // Points [0].X: 1
     01 00 00 00             // Points [0].Y: 1
     0a 00 00 00             // Points [1].X: 10
     02 00 00 00             // Points [1].Y: 2
```

3.14 Dynamic Group

A dynamic group is represented as a relative offset to a location in the data area where the actual value is stored. A dynamic group value in the data area has the exact same structure as a top level message, including a local data area and the ability to carry an unsolicited extension, see Section 3 (page 2). Field values of the group that have variable size are placed in the local data area of the group itself.

Given these definitions:

```
Shape
Rect/7   : Shape -> u32 Wdt, u32 Hgt
Circle/8 : Shape -> u32 Rad
Canvas/9 -> Shape* [] Shapes
```

and a message:

```
@Canvas|Shapes=[@Rect|Wdt=2|Hgt=3;@Circle|Rad=3]
```

it would be encoded as:

```
     48 00 00 00 // Msg size: 72
     09 00 00 00 00 00 00 00 // Type ID for Canvas: 9
     00 00 00 00 // No extension
     04 00 00 00 // Shapes: offset: 4 (1)
                 // -- data area --
(1)  02 00 00 00 // Shapes: size: 2
     08 00 00 00 // Shapes [0]: offset: 8 (2)
     13 00 00 00 // Shapes [1]: offset: 19 (3)
(2)  14 00 00 00 // Dyn grp size: 20
     07 00 00 00 00 00 00 00 // Type ID for Rect: 7
     00 00 00 00 // No extension
     02 00 00 00 // Wdt: 2
     03 00 00 00 // Hgt: 3
(3)  10 00 00 00 // Dyn grp size: 16
     08 00 00 00 00 00 00 00 // Type ID for Circle: 8
     00 00 00 00 // No extension
     03 00 00 00 // Rad: 2
```

4 Message Extensions

A message or a dynamic group may carry unsolicited extension content. The presence of an extension is indicated by a non-zero extension offset that appears in the preamble of the group. The offset is relative to the first byte of the extension offset itself and refers to where in the data area the actual extension value is located. The extension value is represented as a sequence of dynamic groups.

```
Mail/10 -> string Subject, string Body
Trace/11 -> string Hop
```

```
@Mail|Subject=Hello|Body=How are you?|
[@Trace|Hop=local.eg.org;@Trace|Hop=mail.eg.org]
```

Given the schema above, the message would be encoded like this:

```
     80 00 00 00    // Msg size: 128
     0a 00 00 00 00 00 00 00 // Type ID for Mail: 10
     25 00 00 00    // Extension offset: 37 (1)
     08 00 00 00    // Subject: offset: 8 (2)
     0d 00 00 00    // Body: offset: 13 (3)
                    // -- data area --
(2)  05 00 00 00    // Subject: size: 5
     48 65 6c 6c 6f // "Hello"
(3)  0c 00 00 00    // Body: size: 12
     48 6f 77 20 61 // "How a"
     72 65 20 79 6f // "re yo"
     75 3f          // "u?"
(1)  02 00 00 00    // Extension: size: 2
     08 00 00 00    // Extension [0]: offset: 8 (4)
     1f 00 00 00    // Extension [1]: offset: 31 (5)
(4)  20 00 00 00    // Dyn grp size: 32
     0b 00 00 00 00 00 00 00 // Type ID for Trace: 11
     00 00 00 00    // No extension
     04 00 00 00    // Hop: offset: 4 (6)
                    // -- local data area --
(6)  0c 00 00 00    // Hop: size: 12
     6c 6f 63 61 6c // "local"
     2e 65 67 2e 6f // ".eg.o"
     72 67          // "rg"
(5)  1f 00 00 00    // Dyn grp size: 31
     0b 00 00 00 00 00 00 00 // Type ID for Trace: 11
     00 00 00 00    // No extension
     04 00 00 00    // Hop: offset: 4 (7)
                    // -- local data area --
(7)  0b 00 00 00    // Hop: size: 11
     6d 61 69 6c 2e // "mail."
     65 67 2e 6f 72 // "eg.or"
     67             // "g"
```

A decoder that reads the extension groups should skip those with unknown type.

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

A Encoding Grammar

This grammar gives an overview of the encoding structure. However, many key concepts like data area allocation and offsets are not captured here.

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

```
stream ::=
   e
 | dynGroup stream

dynGroup ::=
   length typeId extOff group data

group ::=
   e
 | field group

field ::=
   value
 | nullableValue

nullableValue ::=
   presenceFlag value

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

u8 ::=
   byte

i8 ::=
   byte

u16 ::=
   byte byte

i16 ::=
   byte byte

u32 ::=
   byte byte byte byte

i32 ::=
   byte byte byte byte

u64 ::=
   byte byte byte byte byte byte byte byte

i64 ::=
   byte byte byte byte byte byte byte byte

f64 ::=
   byte byte byte byte byte byte byte byte

bool ::=
   "\x00" | "\x01"

decimal ::=
   exponent mantissa

mantissa ::=
   i64

exponent ::=
   i8

millitime ::=
   i64

nanotime ::=
   i64

date ::=
   i32

timeOfDayMilli ::=
   u32

timeOfDayNano ::=
   u64

inlineString ::=
   shortLength bytes

fixed ::=
   bytes

offset ::=
   u32

length ::=
   u32

shortLength ::=
   u8

presenceFlag ::=
   bool

extOff ::=
   offset

typeId ::=
   u64

data ::=
   e
 | dataItem data

dataItem ::=
   string | dynGroup | sequence

string ::=
   length bytes

sequence ::=
   length items

items ::=
   e
 | value items

bytes ::=
   e
 | byte bytes

byte ::=
   [\x00-\xff]
```

B References

BLINK	http://blinkprotocol.org/spec/BlinkSpec-beta4.pdf
TWOC	http://en.wikipedia.org/wiki/Two's_complement
UTF-8	http://tools.ietf.org/html/rfc3629