Blink Tag Format Specification

beta4 - 2013-06-14

This document specifies a method for encoding messages structured by a Blink schema into a plain text tag-value format. The format is suitable for crafting Blink messages in a format readable by humans while still being machine processable.

Contents

Appendices

1 Overview

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

This specification provides a complementary format called Blink Tag, that will represent messages defined by a Blink schema as plain text. The purpose of this format is to be human readable while still being machine processable.

Possible usages include:

- Crafting of short messages for test purposes when developing new protocols or for inclusion in documentation
- Text logging of messages in applications using Blink
- Creation of text based test data sets
- Simple string based creation of messages inside applications

The Tag format uses self describing type and field names, also called tags, and is line oriented: each message comprises a single line. This makes it suitable for post-processing by generic, text based tools like **`grep`**.

The Tag format provides full fidelity to the core specification: any message encoded in the compact binary format can also be represented by a message in the Tag format as long as the contained types are defined by a schema.

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

Assuming the schema:

```
Hello -> string Greeting
```

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

```
@Hello|Greeting=Hello 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 6 (page 5) for details.

Type names written in a monospaced font like this: **`u64`**, refer to the corresponding value types in the Blink schema language.

3 Format

The following grammar [EBNF] summarizes the Tag format syntax. The full grammar is available here: Appendix A (page 5) .

```
stream      ::= (msg "\n")*
msg         ::= type ("|" fields)? extension?
group       ::= fields?
              | type extension?
              | type "|" fields extension?
fields      ::= field ("|" field)*
field       ::= tag "=" value
value       ::= literal
              | hexList
              | "{" group "}"
              | sequence
sequence    ::= "[" items? "]"
hexList    ::= "[" (hex | " ")* "]"
items       ::= item (";" item)*
item        ::= value | group
extension   ::= "|[" groups? "]"
groups      ::= group (";" group)*
type        ::= "@" name
tag         ::= ncName
name        ::= ncName | cName
cName       ::= ncName ":" ncName
ncName      ::= nameStart nameChar*
nameChar    ::= nameStart | digit
nameStart   ::= [_a-zA-Z]
digit       ::= [0-9]
literal     ::= literalChar*
literalChar ::= [^\x00-\x1f|[]{};#\] | escape
escape      ::= "\" [n|[]{};#\]
              | "\x" twoHex
              | "\u" twoHex twoHex
              | "\U" twoHex twoHex twoHex twoHex
twoHex      ::= hex hex
hex         ::= [0-9a-fA-F]
```

A stream of Tag messages is encoded in the [UTF-8] encoding. Each message comprises a single line. A message line starts with an at sign (`@`), followed by the type name of the message, followed by zero or more fields separated by the bar (`|`) character:

```
@Rect|Width=2|Height=3
```

Each field starts with the name as specified in the schema, followed by an equal sign (`=`) followed by the field value. The fields can appear in any order. It is a weak error (W1) if a field appears more than once.

Optional fields without a value are simply left out in the Tag format:

```
TellTime -> string Timezone?
```

```
@TellTime
@TellTime|Timezone=UTC
```

In the above example the timezone field is optional, and is left out in the first message.

It is a weak error (W2) if a mandatory field is not present in a message. A field is mandatory if it is not marked as optional in the schema.

If the message type was defined in a namespace, then the namespace name should precede the message type name separated by a colon.

Assuming the schema:

```
namespace Draw

Shape ->
  decimal Area?

Circle : Shape ->
  u32 Radius
```

a circle message would look like this:

```
@Draw:Circle|Radius=3|Area=28.3
```

The format of a field value depends on the specified type. The following sections describe the format for each data type. These descriptions together with the corresponding productions in the grammar in Appendix A (page 5) formally define the syntax for each data type.

It is a strong error (S1) if a message line in the stream does not match the production **`line`** in the formal grammar. When matching a message line against the grammar, the declared type of a field should be considered when selecting the appropriate grammar production.

3.1 Integer

An integer is represented in decimal form, possibly with a leading minus sign. Leading zeros are allowed, but a leading plus sign is not allowed.

```
@Point|X=-17|Y=4711
```

It is a weak error (W3) if the spcecified value is not within the range of the declared type for the field.

3.2 String, Binary and Fixed

String, binary and fixed values are logically sequences of bytes at the protocol level. They all share the same syntax in the Tag format, but have different constraints. The following sections specify how to map Tag encoded string values to such byte sequences.

3.2.1 Unicode Strings

A value can be encoded as-is if it can be represented as a valid UTF-8 sequence:

```
@Singer|Name=Robyn # Value: 52 6f 62 79 6e
@Singer|Name=Björk # Value: 42 6a c3 b6 72 6b
```

The control characters **`0x00-0x1f`**, and the reserved characters **`|[]{};#\`** must be escaped. The reserved characters are escaped by preceding them with a backslash. In this example the pipe sign is escaped:

```
@Exec|Command=du -s * \| sort -n
```

An embedded newline character can be represented by the escape sequence **\n**:

```
@BlogPost|Content=<h1>Think Blink</h1>\n<p>...</p>
```

Other control characters can be escaped using the byte or Unicode escape sequences described below.

Any Unicode character can be represented using Unicode escape sequences, ***\uXXXX*** or ***\UXXXXXXXX***, where ***XXXX*** and ***XXXXXXXX*** are a hexadecimal numbers in the range **0x00-0x10ffff**. The hex number represents a Unicode code point. The code point number must be exactly four or eight hexadecimal digits depending on the start of the escape sequence.

Decoding a Unicode escape sequence results in the UTF-8 byte sequence that corresponds to the code point.

> **NOTE**: Surrogate code points, **0xd800-0xdfff**, cannot be represented as valid UTF-8 sequences and must therefore not appear in Unicode escape sequences. For example, **\ud800** is not a valid Unicode escape sequence.

It is a weak error (W4) if a Unicode escape sequence does not represent a valid code point.

The following two messages would appear identical when comparing the bytes of the resulting value:

```
@Calc|Formula=π·r² # Value: cf 80 c2 b7 72 c2 b2
@Calc|Formula=\u03c0\u00b7r\u00b2
```

3.2.2 Binary Strings

A value that is not a valid UTF-8 sequence can be represented using byte escape sequences or it can alternatively be encoded as a *hexadecimal list*.

A byte escape sequence looks like this: ***\xXX***, where ***XX*** is the hexadecimal value of the byte.

Assuming a schema:

```
inetAddr = fixed (4)
Packet ->
  inetAddr Host,
  binary Data
```

a message could be encoded like this:

```
@Packet|Host=\x3e\x6d\x3c\xea|Data=GET / HTTP/1.0\x0d
\n\x0d\n
```

> **NOTE**: The byte and Unicode escape sequences differ in that the former results in a single byte in the decoded value, and the latter in a UTF-8 byte sequence. For example, **\xb2** results in the single byte **0xb2**, and **\u00b2** results in the UTF-8 sequence for the character "superscript two", which is **0xc2 0xb2**.

A hexadecimal list is a sequence of hex digits possibly separated by spaces. The list is enclosed in square brackets (**[]**). The resulting byte sequence is obtained by removing any spaces and translating each pair of hex digits into a byte. It is a strong error (S2) if the number of hex digits is not a multiple of two.

The host address in the previous example would look like this using the hexadecimal list method:

```
@Packet|Host=[3e 6d 3c ea] # Value: 62.109.60.234
```

3.2.3 Constraints

The types based on byte sequences have the following type-specific constraints.

string	The byte sequence must be a valid UTF-8 sequence. If the type has a max size property, then the byte sequence must not be longer than the specified value.
binary	If the type has a max size property, then the byte sequence must not be longer than the specified value.
fixed	The byte sequence must have excatly the same number of bytes as specified in the size property on the type.

Values of all the three types, **string**, **binary** and **fixed** can be represented using any of the methods described above as long as the constraints in this section are not violated. It is a weak error (W5) if one of these constraints is violated.

3.3 Enumeration

An enumeration value is represented by the corresponding symbol name:

```
Color = Red | Green | Blue
Car   -> Color Color
```

Assuming the schema above, the color of a couple of cars would be represented like this:

```
@Car|Color=Blue
@Car|Color=Red
```

It is a weak error (W6) if the value is not the name of a symbol in the corresponding enumeration in the schema.

3.4 Boolean

The Boolean values true and false are encoded as the characters **Y** and **N** respectively:

```
@Logon|KeepAlive=Y
```

3.5 Decimal

A **`decimal`** value is encoded using a standard decimal notation optionally followed by an exponent specifier:

```
@Order|Price=4711.17
@Order|Price=471117E-2
@Order|Price=47.1117E2
```

The three price values all represent the same decimal value:

$$471117 \cdot 10^{-2}$$

It is a weak error (W7) if the specified value cannot be represented by an **`i64`** mantissa and an **`i8`** exponent

3.6 Floating point

A **`f64`** value has two possible representations:

- a decimal number with an optional exponent part as defined for the **`decimal`** type above. In addition, three special values **`Inf`**, **`-Inf`** and **`NaN`** are allowed,
- and a hexadecimal representation of the IEEE 754-2008 bit pattern preceded by a **`0x`** prefix. The leftmost bit has the highest significance and is the sign bit.

```
4711.17
-471117E-2         # -4711.17
0x40b2672b851eb852 # 4711.17
0x7ff0000000000000 # Inf
0xfff0000000000000 # -Inf
0xfff8000000000000 # NaN
```

3.7 Time

A **`millitime`** and **`nanotime`** value is represented as a combined date and time of day in the *basic* or *extended* format as defined in ISO 8601 [TIME]. The separating **`T`** character can be omitted in the basic format, and it can alternatively be replaced with a single space character. The subsecond part follows after a dot (**`.`**), and can be omitted if it is zero. If both the second and subsecond parts are zero, both may be omitted.

If no timezone is specified, then local time is assumed. Other timezones may be specified as defined in ISO 8601.

```
2012-11-20 10:05:30.323115072 # Extended, nanotime
2012-11-20 10:05:30.323       # Extended, millitime
2012-11-20T10:05:30.323       # T as separator
2012-11-20 10:05:30           # Subsecond part is zero
2012-11-20 10:05              # Subminute part is zero

20121120 100530.323           # Basic, millitime
20121120T100530.323           # T as separator
20121120100530                # Omitted separator

2012-11-20 09:05:30Z          # UTC
2012-11-20 10:05:30+01        # One hour ahead of UTC
```

3.8 Date

A **`date`** value is represented in a complete *basic* or *extended* format as defined in ISO 8601 [TIME]:

```
2012-11-20
20121120
```

3.9 Time of Day

A **`timeOfDayMill`** or **`timeOfDayNano`** value is represented in a complete *basic* or *extended* format as defined in ISO 8601 [TIME]. The subsecond part follows after a dot (**`.`**), and can be omitted if it is zero. If both the second and subsecond parts are zero, both may be omitted.

```
10:05:30.323000000
10:05:30.323
100530.323
100530
1005
```

3.10 Static Group

A static group value is represented by encoding the fields of the group. When a static group appears directly as a field value, it must be enclosed in braces (**`{}`**). If the value appears as a sequence item, then the braces are optional.

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

A couple of examples using the above schema:

```
@Rect|Pos={X=3|Y=4}|Width=10|Height=10 # Field value
@Path|Points=[X=1|Y=1;X=10|Y=2]        # Sequence item
@Path|Points=[{X=1|Y=1};{X=10|Y=2}]    # With braces
```

3.11 Dynamic Group

A dynamic group is encoded just as a top level message: a type name following an at sign, followed by zero or more fields. When a dynamic group appears directly as a field value, it must be enclosed in braces (**`{}`**). If the value appears as a sequence item, then the braces are optional.

```
Frame  -> u64 SeqNo, object Payload
Update -> Record* [] Records
Record -> u64 Id
Person : Record -> string Name
Room   : Record -> string Location
```

Assuming the above schema, an imaginary data base update could look like this:

```
@Frame|SeqNo=1|Payload={@Update|Records=[@Person|
Id=1|Name=George;@Room|Id=2|Location=West wing]}
```

It is a weak error (W8) if the specified type name does not name a group in the schema.

3.12 Sequence

A sequence value is represented by zero or more items enclosed in square brackets (**[]**). The items are separated by semicolon (**;**).

```
[1;2;3;4]              # Sequence of integers
[Pale Ale;Lager;Stout] # Sequence of strings
[X=1|Y=2;X=3|Y=4]      # Sequence of static groups
[]                     # Empty sequence
```

4 Message Extensions

In Blink, a message or a dynamic group may carry unsolicited extension content. In the Tag format, an extension appears last in the group and is represented as a sequence field without a tag.

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

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

A decoder reading the Tag format should skip and ignore extensions with unknown types.

5 Comments and Blanks

The Tag format allows comments to appear immediately before the newline character. A comment starts with a pound sign (**#**) and extends to the end of the line. Blank lines, consisting entirely of whitespace and possibly a comment, may appear interleaved with the message lines.

```
# This is a comment followed by a blank line

@Foo|Bar=Baz# This comment ends a message line
@Foo|Bar=Baz # Oops, an extra space added to the value
```

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 line 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.

A Tag Format 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. Nested brackets in a character class are treated literally.

```
stream ::=
   e
 | line
 | line "\n" stream

line ::=
   msg
 | msg comment
 | separator

msg ::=
   dynGroup

dynGroup ::=
   type extension
 | type "|" fields extension

group ::=
   e | fields

fields ::=
   field
 | field "|" fields

field ::=
   tag "=" fieldValue

fieldValue ::=
   value | sequence

value ::=
   integer | string | hexList | enum | bool |
   decimal | f64 | timestamp | date | timeOfDay |
   "{" groupVal "}"

sequence ::=
   "[]"
 | "[" items "]"

items ::=
   item
 | item ";" items

item ::=
   value | groupVal

groupVal ::=
   group | dynGroup

extension ::=
   e
 | "|[]"
 | "|[" dynGroups "]"

dynGroups ::=
   dynGroup
 | dynGroup ";" dynGroups

type ::=
   "@" name

tag ::=
   ncName

integer ::=
   digits
 | "-" digits

string ::=
```

```
   e
 | strChar string

strChar ::=
   [^\x00-\x1f|[]{};#\]
 | escape

escape ::=
   "\" [n|[]{};#\]
 | "\x" twoHex
 | "\u" twoHex twoHex
 | "\U" twoHex twoHex twoHex twoHex

twoHex ::=
   hexDigit hexDigit

hexList ::=
   "[" hexItems "]"

hexItems ::=
   e
 | hexItem hexItems

hexItem ::=
   " " | hexDigit

enum ::=
   ncName

bool ::=
   [YyNn]

decimal ::=
   decNum (exp | "-" decNum exp)

decNum ::=
   digits
 | digits "." digits

exp ::=
   e
 | [Ee] integer

f64 ::=
   decimal | hexNum | "Inf" | "-Inf" | "NaN"

timestamp ::=
   basicDate tsep basicTod basicTz
 | basicDate basicTod basicTz
 | extDate tsep extTod extTz

tsep ::=
   [T\x20]

date ::=
   basicDate | extDate

basicDate ::=
   year two two

extDate ::=
   year "-" two "-" two

year ::=
   digit digit digit digit

two ::=
   digit digit

timeOfDay ::=
   basicTod | extTod

basicTod ::=
   two two subsec
 | two two two subsec

extTod ::=
   two ":" two subsec
 | two ":" two ":" two subsec

subsec ::=
   e
 | "." digits

basicTz ::=
   tz
 | sign two two

extTz ::=
   tz
 | sign two ":" two

tz ::=
   e
 | "Z"
 | sign two

sign ::=
   [-+]

name ::=
   ncName | cName

cName ::=
   ncName ":" ncName

ncName ::=
   nameStartChar nameChars

nameChars ::=
   nameChar
 | nameChar nameChars

nameChar ::=
   nameStartChar | digit

nameStartChar ::=
   [_a-zA-Z]

digits ::=
   digit
 | digit digits

digit ::=
   [0-9]

hexNum ::=
   "0x" hexDigits

hexDigits ::=
   hexDigit
 | hexDigit hexDigits

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

separator ::=
   blank
 | blank comment

comment ::=
   "#" restOfLine

restOfLine ::=
   e
 | [\n] restOfLine

blank ::=
   e
 | [\x20\x09] blank
```

B References

BLINK	http://blinkprotocol.org/spec/BlinkSpec-beta4.pdf
EBNF	http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation
TIME	http://dotat.at/tmp/ISO_8601-2004_E.pdf
UTF-8	http://tools.ietf.org/html/rfc3629