FIX/Blink Specification

beta1 - 2013-06-14

This document specifies how to encode FIX, as is, on Blink in an interoperable way. The specification leverages the Blink message schema exchange protocol to support ad hoc message extensions.

Contents

1 **Overview** . . . 1

2 **Schema Impedance Discussion** . . . 1

3 **Schema** . . . 1

- 3.1 Namespace . . . 1
- 3.2 Message Base . . . 2
- 3.3 Message . . . 2
- 3.4 Field . . . 2
- 3.5 Component Block . . . 2
- 3.6 Repeating Group . . . 2
- 3.7 Enumeration . . . 2
  - 3.7.1 Numeric . . . 2
  - 3.7.2 Single Character . . . 2
  - 3.7.3 String . . . 2
- 3.8 Excluded Fields . . . 3
  - 3.8.1 BeginString . . . 3
  - 3.8.2 BodyLength . . . 3
  - 3.8.3 MsgType . . . 3
  - 3.8.4 Repeating Group Length Fields . . 3
- 3.9 Type Mapping . . . 3
- 3.10 Custom Tags . . . 3
- 3.11 Message Type Identifiers . . . 3
- 3.12 Annotations . . . 3

Appendices

A **Examples** . . . 4

B **References** . . . 5

1 Overview

This document outlines how FIX can be layered on the Blink protocol in an interoperable way, without making any modifications to the definition of FIX messages or the FIX session layer. Encoding FIX, as is, on Blink yields compactness and performance benefits over traditional text based tag/value encoding.

A potential refresh of the FIX application layer targeting further efficiency benefits is outside the scope of this document.

2 Schema Impedance Discussion

FIX is structured into relatively few message types where each type can represent a number of scenarios by mixing together different sets of fields per scenario. This structure goes hand in hand with a tagged representation.

In the Blink encoding, absent optional fields occupy one byte each. Trailing absent optional fields can be left out, causing no encoding overhead, but it may be difficult to order fields in an optimal way. Blink therefore incurs an overhead in a structure having a large number of optional fields.

The philosophy that led to Blink, and for which its simple design is optimized, is to define one message type per major use scenario. From this follows that there are relatively few optional fields in each message definition. The design philosophy for supporting flexible and emergent use cases is to define new message types in the same or a new Blink message schema, whereas the philosophy of FIX has been to add new optional fields to existing broad message types when applicable. This gap precludes what would otherwise seem to be a good idea: to make an automatic one-to-one mapping of the FIX repository onto a corresponding Blink message schema.

Instead, the FIX model of having a large amount of optional fields always available needs to be mapped onto a model where unused fields are left out.

How this mapping is performed is outside the scope of this document. Typically, specifications of FIX rules of engagement can be used as the source for the mapping into a definition in Blink message schema format. Each venue, service bureau, sell-side, or other entity that provides a FIX service has a formal or informal specification of their rules of engagement, and that can form the basis for the message schema to be defined in that particular setting.

It should be possible to specify this subsetting in a data format that allows tools to automatically generate the resulting sub schemas in Blink format from FIX repository schema files.

3 Schema

Most of the samples below are taken from the CME iLink rules of engagement.

3.1 Namespace

The namespace for the schema is the FIX version being defined. A rules of engagement specification based on FIX 4.2 has the namespace:

```
namespace Fix42
```

The namespace for FIXT 1.1 is **FixT11**.

3.2 Message Base

Fields from the standard header are placed in a group called **Message**:

```
Message ->
  bool PossDupFlag, ...
```

3.3 Message

A message is named to match the original FIX name. Every message inherits from **Message** to include header fields:

```
ExecutionReport : Message ->
  bool AggressorIndicator?, ...
```

3.4 Field

A field is named to match the original FIX name:

```
bool AggressorIndicator?
```

A field in Blink can be mandatory or optional just as in FIX. However, when constructing the Blink schema for a particular set of rules of engagement, a field that is marked as optional in the original specification may be mapped to a mandatory field if it is always present in practice.

3.5 Component Block

FIX component blocks are groups of fields that can be referenced by a component name from within a message definition.

A component block is mapped to a static subgroup type in Blink.

3.6 Repeating Group

In the FIX data model a repeating group lacks a real field name. For the purpose of this mapping, a field name is created by removing the 'No' prefix of the corresponding length field: the field name of 'NoPartyIDs' becomes 'PartyIDs'.

> **NOTE**: The exception to this rule is the 'LinesOfText' sequence in the News and Email message types, whose field name will be identical to the repeating group name.

A repeating group is mapped to a sequence type where the item type is a group compatible with the original structure. The content of the original repeating group depends on the FIX version being mapped. It is either

- a component block reference - in this case the item type simply becomes a reference to the group that corresponds to the component block;
- or a set of inline fields - a matching group is created by concatenating the name of the group, component or enclosing repeating group where the repeating group appears with the mapped field name.

The length field itself is not mapped since the sequence type in Blink has an intrinsic length value.

The content of the sequence is placed in a group:

```
PartiesPartyIDs ->
  string PartyID?,
  PartyIDSource PartyIDSource?,
  PartyRole PartyRole?,
  PtysSubGrpPartySubIDs [] PartySubIDs?
```

```
PartiesPartyIDs []  PartyIDs?, ...
```

3.7 Enumeration

An enumeration in FIX is mapped to an enumeration in Blink unless the actual used set of valid values is open ended. If it is open ended, then mapping of the underlying type is used instead.

> **NOTE**: Rules of engagement may constrain an otherwise open ended enumeration so that it becomes finite and can be mapped to a Blink enumeration.

An enumeration based on strings is by default not mapped to an enumeration in Blink to minimize the need for lookup tables in translating implementations. However, rules of engagement may specify that string enumerations are mapped to native Blink enumerations too.

FIX enumerations can be categorized into three groups: numeric, single character, and string. The following sections describe the mappings in the three different cases.

3.7.1 Numeric

A numeric enumeration is mapped directly to a Blink enumeration where each symbol has the same value as in the original:

```
BusinessRejectReason = Other/0 | UnknownId/1
 | UnknownSecurity/2 | UnsupportedMessage/3
```

3.7.2 Single Character

A single character enumeration maps to a Blink enumeration where each symbol has the corresponding character code as the value:

```
CxlRejResponseTo = OrderCancelRequest/49
 | OrderCancelReplaceRequest/50
```

3.7.3 String

By default a string enumeration maps to a plain string field in Blink:

```
string BenchmarkCurveName?
```

The rules of engagement may specify that also a string based enumeration is mapped to a native Blink enumeration and must then specify how the FIX values map to corresponding Blink enum symbols.

3.8 Excluded Fields

The **`BeginString`**, **`BodyLength`**, **`MsgType`** and **`Checksum`** fields are left out as they are artefacts of the tag/value encoding and have a different representation or no meaning at all in the Blink encoding.

3.8.1 BeginString

The begin string follows implicitly from the mapped group type.

3.8.2 BodyLength

Is not applicable when a message is encoded over Blink.

3.8.3 MsgType

The msg type follows implicitly from the mapped group type.

3.8.4 Repeating Group Length Fields

The repeating group length fields are eliminated as sequences are native in Blink and include the sequence size.

3.9 Type Mapping

This is the general default mapping from the FIX *base type* into Blink:

- *Boolean* - **bool**
- *int* - **i32**
- *String* - **string**
- *char* - **u8**
- *float* - **decimal**
- *XML data* - **string(text/xml)**

The following list overrides the above default mapping for some type classes, to extend the value range:

- *All Seqnum* - **u64**

The following list overrides the above general mapping for some derived types, to avoid unnecessary use of **String**:

- *UTCTimestamp* - **millitime**
- *LocalMktDate* - **date**
- *UTCDateOnly* - **date**
- *UTCTimeOnly* - **timeOfDayMilli**

3.10 Custom Tags

The Blink message extension mechanism is used to add dynamic content to messages. A receiver is able to skip new content.

A sender defining content on the fly, for instance by adding a custom tag, is required to define the new content using Blink Schema Exchange.

A receiver that processes the Blink Schema Exchange messages will be able to process any subsequent content, including fields that have not previously been defined.

In order for implementations to be interoperable, a convention on how the extension mechanism is applied must be established. This document specifies that custom tags are placed in one or more groups and that the naming of these groups has no meaning. The implication of the presence of these groups is that their content belong in the original message as custom tags.

3.11 Message Type Identifiers

The Blink encoding requires a numerical identifier for each message type. The rules of engagement must specify the actual identifiers to be used. The Blink schema syntax natively provides two ways of specifying type identifiers, inline or out-of-line. Identifiers can be specified as integers or as a hex numbers.

Inline type identifiers are specified after a slash following the message name in the schema:

```
ExecutionReport/0x38 -> ...
NewOrderSingle/0x44 -> ...
```

Out-of-line type identifiers are specified as incremental annotations which allow them to appear in locations separate from the message definitions:

```
Fix42:ExecutionReport <- 0x38
Fix42:NewOrderSingle <- 0x44
```

3.12 Annotations

Blink schemas can be annotated with auxiliary information useful to translating implementations and for human consumption. The Blink schema format supports both inline and out-of-line annotations, called *incremental annotations*.

It is recommended, but not required, that messages and fields are annotated with their corresponding identifiers in the tag/value encoding.

If fields are annotated inline with tag numbers, then they should be specified after a slash following the field name:

```
string Text/58
```

If a type definition is used, then the tag number kan be specified after a slash there:

```
Text/58 = string
```

If messages are annotated inline with the corresponding **`MsgType`** value it should be placed in an annotation named fix:msgType:

```
@fix:msgType="A"
Logon -> ...
```

Incremental annotations allow annotations to be specified in a location separate from the schema components they apply to. The location could be later in the same schema file, but it could also be in a separate schema file altogether:

```
Fix42:Logon <- @fix:msgType="A"
Fix42:Logout.Text <- 58
```

A Examples

The following example shows how a **NewOrderSingle**, based on the CME iLink rules of engagement, can be represented in a Blink schema.

```
namespace Fix42

@fix:msgType="D"
NewOrderSingle/0x44 : Message ->
  bool ManualOrderIndicator/1028,
  string ClOrdID/11,
  string Account/1,
  HandlInst HandlInst/21,
  string Symbol/55,
  string SecurityDesc/107,
  Side Side/54,
  OrdType OrdType/40,
  CustomerOrFirm CustomerOrFirm/204,
  CtiCode CtiCode/9702,
  decimal OrderQty/38,
  millitime TransactTime/60,
  CustOrderHandlingInst CustOrderHandlingInst/1031?,
  decimal Price/44?,
  TimeInForce TimeInForce/59?,
  string SecurityType/167?,
  NewOrderSingleAllocs [] Allocs/78?,
  date ExpireDate/432?,
  OpenClose OpenClose/77?,
  string CorrelationClOrdID/9717?,
  string GiveUpFirm/9707?,
  string CmtaGiveupCD/9708?,
  string OmnibusAccount/9701?,
  decimal MinQty/110?,
  decimal StopPx/99?,
  decimal MaxShow/210?

NewOrderSingleAllocs ->
  string AllocAccount/79

Message ->
  string SenderCompID/49,
  string TargetCompID/56,
  string SenderSubID/50,
  string TargetSubID/57,
  u64 MsgSeqNum/34,
  millitime SendingTime/52,
  u64 LastMsgSeqNumProcessed/369,
  string SenderLocationID/142?,
  millitime OrigSendingTime/122?,
  bool PossDupFlag/43?,
  bool PossResend/97?

TimeInForce =
  Day/48 | GoodTillCancel/49 | AtTheOpening/50 |
  ImmediateOrCancel/51 | FillOrKill/52 |
  GoodTillCrossing/53 | GoodTillDate/54

Side =
  Buy/49 | Sell/50 | BuyMinus/51 | SellPlus/52 |
  SellShort/53 | SellShortExempt/54 |
  Undisclosed/55 | Cross/56 | CrossShort/57

HandlInst =
  AutoExecOrderPrivateNoBrokerIntervention/49 |
  AutoExecOrderPublicBrokerInterventionOk/50 |
  ManualOrderBestExecution/51

OpenClose =
  Close/67 | Open/79

CtiCode =
  OwnAccount/49 | ClearingMemberHouseAccount/50 |
  OtherFloorBrokerTraderAccount/51 |
  OtherNonFloorAccount/52
```

```
OrdType =
  Market/49 | Limit/50 | Stop/51 | StopLimit/52 |
  MarketOnClose/53 | WithOrWithout/54 |
  LimitOrBetter/55 | LimitWithOrWithout/56 |
  OnBasis/57 | OnClose/65 | LimitOnClose/66 |
  ForexMarket/67 | PreviouslyQuoted/68 |
  PreviouslyIndicated/69 | ForexLimit/70 |
  ForexSwap/71 | ForexPreviouslyQuoted/72 |
  Funari/73 | Pegged/80

CustomerOrFirm =
  Customer/0 | Firm/1

CustOrderHandlingInst =
  PhoneSimple/65 | PhoneComplex/66 |
  FcmProvidedScreen/67 | OtherProvidedScreen/68 |
  ClientProvidedPlatformControlledByFcm/69 |
  ClientProvidedPlatformDirectToExchange/70 |
  FcmApiOrFix/71 | AlgoEngine/72 |
  PriceAtExecution/74 | DeskElectronic/87 |
  DeskPit/88 | ClientElectronic/89 | ClientPit/90
```

B References

BLINK http://blinkprotocol.org/spec/BlinkSpec-beta4.pdf

EXCH http://blinkprotocol.org/spec/BlinkSchemaExchangeSpec-beta4.pdf