Encoding FIX using Google Protocol Buffers

Release Candidate 3

Technical Standard Proposal

Nov 19, 2018

Rev 5

Proposal Status: Public Review

For Global Technical Committee Governance Internal Use Only

Submission Date		Control Number	
Submission Status		Ratified Date	
Primary Contact Person		Release Identifier	

DISCLAIMER

Table of Contents

Document History

Revision	Date	Author(s)	Revision Comments
1	Aug 3, 2018	Greg Malatestinic, Jordan & Jordan Sara Rosen, CME	Initial draft
5	Nov 19, 2018	Greg Malatestinic, Jordan & Jordan Sara Rosen, CME	Updated to reflect TZ time data types and committee feedback

1 Introduction

The Google Protocol Buffer encoding utilizes existing industry standard encodings and applies them to the FIX domain. Google Protocol Buffers is a mature binary encoding which can be applied "out of the box" to support FIX language semantics.

The richness of this technology supports multiple solutions to mapping the existent FIX data types to a binary protocol. In the interest of standardization, a specification of a normative encoding of the FIX data types in protocol buffers for interoperable exchange of financial data over FIX has been developed.

The purpose of RC3 of the Google Protocol Buffers Encoding specification is to address the community feedback received during the public comment period on the RC2 proposal. Areas addressed include enhancements for efficiency, ease of use, and automation, as well as considerations relating to the latest protobuf release from Google.

This document describes the changes to the specification that were adopted based on this feedback.

1.1 Authors

Name	Affiliation	Contact	Role
Greg Malatestinic	Jordan & Jordan	greg.malatestinic@jandj.com	GPB co-lead
Sara Rosen	CME	sara.rosen@nex.com	GPB co-lead

2 Requirements

2.1 Business Requirements

This proposal is intended to introduce changes adopted since the previous release candidate, RC2, was made available to the public. Enhancements have been introduced for improved efficiency and ease of use, and to align with the changes introduced by Google's proto 3 release in 2017.

2.2 Technical Requirements

This section discusses enhancements to the GPB mapping standard since Release Candidate 2.

2.2.1 New Supporting GPB Data Types

The previous release candidate, RC2, included several supporting GPB message types. The inclusion of these message types provides an efficient binary representation of fix datatypes such as timestamps and decimal prices which are encoded in classic FIX as ascii strings. In RC3, we have modified and expanded the set of supporting GPB message types for the following datatypes:

Decimal32 and Decimal64

In RC2, each numeric FIX field was mapped to a set of two GPB fields, a mantissa value and an exponent value. The exponent field was identified by concatenating the "_exponent" suffix to the field name.

The following example shows how the numeric fields MinQty and Price were defined in RC2:

```
// RC2 example
message NewOrderSingle {
      // …
      optional sfixed64 min_qty = 10;
      optional sfixed32 min_qty_exponent = 11;
      optional sfixed64 price = 12;
      optional sfixed32 price_exponent = 13;
      // …
}
```

In RC3, the mantissa and exponent are encapsulated into a single GPB decimal message type. This allows a numeric field to be defined within a GPB message with just a single field declaration. It also removes the need to rely on field naming conventions to associate the mantissa and exponent of a single FIX field.

To accommodate varying ranges of values and precisions, 32 and 64-bit versions of the message type have been specified. Accordingly, two new GPB message types, Decimal32 and Decimal64, have been introduced in RC3.

The following example shows how the same two numeric fields are defined in RC3 using the new message types, Decimal32 and Decimal64:

```
// RC3 example
message Decimal32 {
      optional sfixed32 mantissa = 1;
      optional sfixed32 exponent = 2;
}

message Decimal64 {
      optional sfixed64 mantissa = 1;
      optional sfixed32 exponent = 2;
}

message NewOrderSingle {
      // …
      optional Decimal32 min_qty = 10;
      optional Decimal64 price = 11;
      // …
}
```

Note that the above approach involves explicit inclusion of the decimal precision within the GPB message. This supports evolution to higher levels of precision without breaking backward compatibility.

For implementations that do not require this flexibility, a decimal field may be mapped to a single integer (the mantissa), and the precision can be implicitly communicated via the encoding attributes of the field or the specification's Rules of Engagement document. Parsing latency is thus reduced by keeping the message flat.

The Google Protocol Buffer HPWG subcommittee recommends utilizing the explicit Decimal representation based on its objectives for message transparency, ease of use, and backward compatibility. Nevertheless, implementations whose primary focus is parsing speed may elect to encode decimals as fixed-point integers, and this approach also complies with the GPB encoding standard.

Timestamp and TimeOnly

In RC2, a FIX UTCTimestamp was represented by an integer offset, an epoch and a time unit. These were mapped to a single GPB scalar field using the custom options, "epoch" and "time_unit". The epoch and time unit would not be sent over the wire. Rather, they would be specified as metadata in the proto file and in an organization's Rules of Engagement document. Only the offset value would be transmitted. The following example shows how a timestamp field was defined in RC2:

```
// RC2 example
message OrderCancelReject {
      // …
      optional sfixed64 transact_time = 15 [(epoch)=EPOCH_UNIX,
            (time_unit)=TIME_UNIT_MILLISECONDS];
      // …
}
```

The disadvantages of this mechanism are two-fold:

1. Automated toolsets cannot process the data field without loading the values of the encoding attributes embedded in the proto.
2. Timestamps could not evolve for increasing levels of granularity without breaking backward combability.

RC3 utilizes a GPB message type, Timestamp, to represent a FIX UTCTimestamp field. Timestamp is a standard Proto3 message type, which has been back-ported to Proto2. In this message type, there is no need to define the time unit as metadata because the fields explicitly provide for nanosecond precision. The Timestamp message type also standardizes on the UNIX epoch rather than providing support for other epochs.

The following example shows how timestamps are defined in RC3:

```
// RC3 example
message Timestamp {
      optional int64 seconds = 1;  // seconds offset to UNIX epoch
      optional int32 nanos = 2;    // positive nano offset to seconds
}
message OrderCancelReject {
      // …
      optional Timestamp transact_time = 15;
      // …
}
```

Note that, in Timestamp, the 'seconds' field represents signed seconds of time since the UNIX epoch, Jan 1, 1970 00:00:00 UTC, while the 'nanos' field always represents positive fractions of a second at nanosecond resolution. Even when the 'seconds' value is negative, the fractional nanoseconds must still have a non-negative value that counts forward in time as a positive offset to the second.

FIX UTCTimeOnly datatypes are likewise defined using the new proto3 TimeOnly message type. The TimeOnly message type has the same field definition as Timestamp:

```
// RC3 example
message TimeOnly  {
      optional int64 seconds = 1;     // offset to previous midnight
      optional int32 nanos = 2;       // positive offset to seconds
}
```

However, with TimeOnly, the 'seconds' field represents seconds of time since the previous midnight in UTC, 00:00:00 UTC.

TzTimeOnly and TzTimestamp

FIX supports time-zoned times using the TZTimeOnly and TZTimestamp data methods. These contain a string representation of the time/timestamp together with a constant signed offset from UTC.

The binary encoding of this data type utilizes the UTC Timestamp/TimeOnly representation, with the addition of two signed integer fields for the hourly & minute offset from UTC time zone.

TzTimeOnly represents the FIX TZTimeOnly datatype of format HH:MM[:SS][Z | [+ | - hh[:mm]]].

The UTC portion of the timestamp is represented using the same fields as the GPB TimeOnly message type.

```
message TzTimeOnly
{
      optional int64 seconds = 1; // number of seconds since midnight
      optional int32 nanos = 2;   // positive nanosecond offset to seconds
      optional sint32 hour_offset = 3; // signed hourly offset to UTC
      optional sint32 minute_offset = 4; // signed minute offset to UTC
}
```

TzTimestamp represents a date-time. In this case, the seconds field represents the number of seconds since Unix epoch, with the addition of the signed offsets from UTC time.

```
message TzTimestamp
{
      optional int64 seconds = 1; // number of seconds since Unix epoch
      optional int32 nanos = 2;   // positive nanosecond offset to seconds
      optional sint32 hour_offset = 3; // signed hourly offset to UTC
      optional sint32 minute_offset = 4; // signed minute offset to UTC
}
```

Note that when a negative timezone offset to UTC contains both hours and minutes, both the hourOffset and minuteOffset fields must be set to negative values.

LocalMarketTime

Lastly, we introduce a binary representation of the elements of the FIX LocalMktTime datatype which was introduced in FIX 5.0 SP2 EP161. Though it is not included in the FIX datatype definition, nanoseconds have been included to provide additional granularity.

```
message LocalMarketTime {
   optional int32 hours = 1; // hour in local time zone, using 24-hour clock
   optional int32 minutes = 2; // minutes in local time zone
   optional int64 seconds = 3; // seconds in local time zone
   optional int32 nanos = 4; // positive nanosecond offset to seconds
}
```

2.2.2 Default enumeration values

A default value for enumerated types has been added. This is to avoid misinterpreting data if fields are un-initialized.

For example, in RC2, the IOIQty enumeration was mapped as follows:

```
// RC2 example
enum IOIQtyEnum {
    IOI_QTY_SMALL = 0                   [(fix.enum)="S"];
    IOI_QTY_MEDIUM = 1                  [(fix.enum)="M"];
    IOI_QTY_LARGE = 2                   [(fix.enum)="L"];
    IOI_QTY_UNDISCLOSED_QUANTITY = 3    [(fix.enum)="U"];
}
```

In RC3, the enumeration is mapped as follows:

```
// RC3 example
enum IOIQtyEnum {
    IOI_QTY_UNSPECIFIED = 0;
    IOI_QTY_SMALL = 1                   [(fix.enum_value)="S"];
    IOI_QTY_MEDIUM = 2                  [(fix.enum_value)="M"];
    IOI_QTY_LARGE = 3                   [(fix.enum_value)="L"];
    IOI_QTY_UNDISCLOSED_QUANTITY = 4    [(fix.enum_value)="U"];
}
```

2.2.3 Enum encoding attribute

RC2 support for the custom option, “enum”, provided for the definition of enumeration values which are transmitted over the wire.

The following listing shows how the custom option “enum” was defined in RC2 using the field “AdvSide” as an example:

```
// RC2 example
enum AdvSideEnum {
      ADV_SIDE_BUY = 0   [(enum)="B", (enum_added)=VERSION_FIX_2_7];
      ADV_SIDE_CROSS = 1 [(enum)="X", (enum_added)=VERSION_FIX_2_7];
      ADV_SIDE_SELL = 2  [(enum)="S", (enum_added)=VERSION_FIX_2_7];
      ADV_SIDE_TRADE = 3 [(enum)="T", (enum_added)=VERSION_FIX_2_7];
}
```

However, when Java code is generated by the GPB compiler, protoc, the name of the custom option “enum” can conflict with the Java reserved word “enum”. In RC3 the name of the GPB custom option has been changed to “enum_value” to avoid this conflict.

The following listing shows how the renamed custom option is defined in RC3, using the same field as an example. (Also notice the addition of the default value.)

```
// RC3 example
enum AdvSideEnum {
      ADV_SIDE_UNSPECIFIED = 0;
      ADV_SIDE_BUY = 1   [(enum_value)="B", (enum_added)=VERSION_FIX_2_7];
      ADV_SIDE_CROSS = 2 [(enum_value)="X", (enum_added)=VERSION_FIX_2_7];
      ADV_SIDE_SELL = 3  [(enum_value)="S", (enum_added)=VERSION_FIX_2_7];
      ADV_SIDE_TRADE = 4 [(enum_value)="T", (enum_added)=VERSION_FIX_2_7];
}
```

2.2.4 Time_unit and Epoch Encoding Attributes

RC2 provided support for the encoding attributes, “time_unit” and “epoch” for timestamp encoding. With RC3, these encoding attributes have been made moot by the introduction of the supporting types for time-related data, and, accordingly, support for them has been discontinued.

2.2.5 Union types

In RC2, FIX fields defined with a Union data type were mapped to a message type where each field of the message reflected one of the optional representations. However, there was no protection to ensure that at most a single value was transmitted.

The following example shows how a FIX field with a Union data type was defined in RC2:

```
// RC2 example
enum EventTypeEnum {
      EVENT_TYPE_CALL = 0            [(fix.enum)="2"];
      EVENT_TYPE_OTHER = 1           [(fix.enum)="99"];
      EVENT_TYPE_PUT = 2             [(fix.enum)="1"];
      // …
      EVENT_TYPE_SWAP_START_DATE = 19     [(fix.enum)="8"];
}

message EventTypeUnion {
      optional EventTypeEnum event_type = 1;
      optional sfixed64 event_type_sfixed64 = 2;
}

message EvntGrp {
      // …
      optional string event_text = 4;
      optional EventTypeUnion event_type = 5;
      optional sfixed64 event_time = 6;
}
```

As of RC3, Union types are defined in-line using GPB's new "oneof" construct.

A oneof expression consists of the keyword "oneof" followed by the oneof name, followed by several fields encapsulated in a block using open/close brackets. Oneof fields share memory, and at most one field can be set at the same time. For RC3, a oneof expression is used whenever a FIX field definition includes the "unionDataType" attribute.

The following example shows the same definition in RC3:

```
// RC3 example
enum EventTypeEnum {
      EVENT_TYPE_CALL = 0            [(fix.enum)="2"];
      EVENT_TYPE_OTHER = 1           [(fix.enum)="99"];
      EVENT_TYPE_PUT = 2             [(fix.enum)="1"];
      // …
      EVENT_TYPE_SWAP_START_DATE = 19     [(fix.enum)="8"];
}

message EvntGrp {
      // …
      optional string event_text = 4;
      oneof event_type_union {
            EventTypeEnum event_type = 5;
            fixed32 event_type_reserved100plus = 6;
      }
      optional sfixed64 event_time = 7;
}
```

2.3 User Guide

The "Encoding FIX using Google Protocol Buffers User Guide" has been updated to reflect the above RC3 mapping enhancements.

In addition, the User Guide provides an overview of the changes between proto2 and proto3, and their implications for FIX mapping. Primary focus is on the changes for default field handling and detection of field presence. Guidelines for writing proto2 specifications which will be compatible with proto3 implementations have also been provided.

3 Issues and Discussion Points

3.1 Questions to be decided beyond RC3

4 References

Reference	Version	Relevance	Normative
None			

5 Relevant and Related Standards

Standard	Version	Reference Location
Protocol Buffers v2 Language Specification	2	https://developers.google.com/protocol-buffers/docs/reference/proto2-spec
Protocol Buffers v3 Language Specification	3	https://developers.google.com/protocol-buffers/docs/reference/proto3-spec

6 Intellectual Property Disclosure

Related Intellection Property	Type of IP (copyright, patent)	IP Owner	Relationship to proposed standard

7 Definitions

Term	Definition

8 Project Milestone

8.1 Release Candidate 3 Deliverables

These artifacts will be delivered as Release Candidate 3 and will be displayed in the Tech/Specs section of FIX Trading Community website as well as in GitHub project FIXTradingCommunity/fix-protocol-buffers:

- The technical specification is a separate document "Encoding FIX using Google Protocol Buffers – Release Candidate 3".
- A user guide is provided as a separate document "Encoding FIX using Google Protocol Buffers – Release Candidate 3 - User Guide". This document provides an overview of how to represent FIX data in the protocol buffers language.
- A sample protobuf listing covering all FIX 5.0 SP2 components and messages generated from the FIX 2010 Unified Repository.

8.2 Roadmap