logoFIX Simple Binary Encoding

RC2

**RELEASE CANDIDATE 1: MAY 28, 2013**

**THIS DOCUMENT IS A RELEASE CANDIDATE FOR A PROPOSED FIX TECHNICAL STANDARD. A RELEASE CANDIDATE HAS BEEN APPROVED BY THE GLOBAL TECHNICAL COMMITTEE AS AN INITIAL STEP IN CREATING A NEW FIX TECHNICAL STANDARD. POTENTIAL ADOPTERS ARE STRONGLY ENCOURAGED TO BEGIN WORKING WITH THE RELEASE CANDIDATE AND TO PROVIDE FEEDBACK TO THE GLOBAL TECHNICAL COMMITTEE AND THE WORKING GROUP THAT SUBMITTED THE PROPOSAL. THE FEEDBACK TO THE RELEASE CANDIDATE WILL DETERMINE IF ANOTHER REVISION AND RELEASE CANDIDATE IS NECESSARY OR IF THE RELEASE CANDIDATE CAN BE PROMOTED TO BECOME A FIX TECHNICAL STANDARD DRAFT.**

High Performance Working Group

October 2013

© Copyright 2013 FIX Protocol Limited

Contents

[1 Introduction 5](#_Toc369262063)

[1.1 Binary type system 5](#_Toc369262064)

[1.2 Design principles 5](#_Toc369262065)

[1.3 Message schema 5](#_Toc369262066)

[1.4 Glossary 5](#_Toc369262067)

[1.5 Documentation 6](#_Toc369262068)

[1.6 References 7](#_Toc369262069)

[2 Field Encoding 7](#_Toc369262070)

[2.1 Field aspects 7](#_Toc369262071)

[2.2 FIX data type summary 8](#_Toc369262072)

[2.3 Common field schema attributes 11](#_Toc369262073)

[2.4 Integer encoding 11](#_Toc369262074)

[2.5 Decimal encoding 14](#_Toc369262075)

[2.6 Float encoding 16](#_Toc369262076)

[2.7 String encodings 17](#_Toc369262077)

[2.8 MonthYear encoding 21](#_Toc369262078)

[2.9 Date and time encoding 22](#_Toc369262079)

[2.10 Local date encoding 23](#_Toc369262080)

[2.11 Local time encoding 23](#_Toc369262081)

[2.12 Enumeration encoding 25](#_Toc369262082)

[2.13 Multi-value choice encoding 27](#_Toc369262083)

[2.14 Field value validation 29](#_Toc369262084)

[3 Message Structure 30](#_Toc369262085)

[3.1 Session Protocol and Message Frame 30](#_Toc369262086)

[3.2 SBE Message Header 30](#_Toc369262087)

[3.3 Message Body 32](#_Toc369262088)

[3.4 Repeating Groups 33](#_Toc369262089)

[3.5 Sequence of message body elements 37](#_Toc369262090)

[3.6 Message structure validation 37](#_Toc369262091)

[4 Message Schema 38](#_Toc369262092)

[4.1 XML namespace 38](#_Toc369262093)

[4.2 Name convention 38](#_Toc369262094)

[4.3 Root element 38](#_Toc369262095)

[4.4 Data encodings 39](#_Toc369262096)

[4.5 Message template 48](#_Toc369262097)

[4.6 Field attributes 49](#_Toc369262098)

[4.7 Repeating group schema 51](#_Toc369262099)

[4.8 Schema validation 52](#_Toc369262100)

[4.9 Example message schemas 53](#_Toc369262101)

[4.10 Reserved element names 55](#_Toc369262102)

[5 Schema Extension Mechanism 56](#_Toc369262103)

[5.1 Objective 56](#_Toc369262104)

[5.2 Message schema features for extension 56](#_Toc369262105)

[5.3 Wire format features for extension 57](#_Toc369262106)

[5.4 Message schema extension example 57](#_Toc369262107)

[6 Usage Guidelines 59](#_Toc369262108)

[6.1 Identifier encodings 59](#_Toc369262109)

[7 Examples 60](#_Toc369262110)

[7.1 Flat, fixed-length message 60](#_Toc369262111)

[7.2 Message with repeating group 62](#_Toc369262112)

**Document History**

| Revision | Date | Author | Revision comments |
| --- | --- | --- | --- |
| 0.9.2 RC1 | April 1, 2013 | Don Mendelson  CME Group | Promoted to Release Candidate 1 for public release. |
|  | April 1May 28, 2013 | Don Mendelson  CME Group | Add element to message for multi value choice encoding. |
|  | June 19, 2013 | Don Mendelson  CME Group | Corrected table of values for decimal data type. |
|  | August 5, 2013 | Don Mendelson  CME Group | Schema extension mechanism added. ValueRef attribute for constant of enumeration value. |
|  | August 19, 2013 | Don Mendelson  CME Group | Removed signature from message header. |
|  | Sept. 11, 2013 | Don Mendelson  CME Group | Clarified “template” terminology, added note about MsgType and changed constant string example to not use MsgType. |
|  | Sept. 18, 2013 | Don Mendelson  CME Group | Added definition of mantissa and exponent, clarified that alternate encodings are required, not recommended. |
|  | Sept. 23, 2013 | Don Mendelson  CME Group | Float encoding added of IEEE floating point numbers |
|  | Oct. 11, 2013 | Don Mendelson  CME Group | Updated XML schema and wire format for variable length data and repeating groups. Added example section. Numerous corrections. |

# Introduction

FIX Simple Binary Encoding (SBE) targets high performance trading systems. It is optimized for low latency of encoding and decoding while keeping bandwidth utilization reasonably small. For compatibility, it is intended to represent all FIX semantics.

This encoding specification describes the wire protocol for messages. Thus, it provides a standard for interoperability between communicating parties. Users are free to implement the standard in a way that best suits their needs.

The encoding standard is complimentary to other FIX standards for session protocol and application level behavior.

## Binary type system

In order to support traditional FIX semantics, all the documented field types are supported. However, instead of printable character representations of tag-value encoding, the type system binds to native binary data types, and defines derived types as needed.

The binary type system has been enhanced in these ways:

* Provides a means to specify precision of decimal numbers and timestamps, as well as valid ranges of numbers.
* Differentiates fixed-length character arrays from variable-length strings. Allows a way to specify the minimum and maximum length of strings that an application can accept.
* Provides a consistent system of enumerations, Boolean switches and multiple-choice fields.

## Design principles

The message design strives for direct data access without complex transformations or conditional logic. This is achieved by:

* Usage of native binary data types and simple types derived from native binaries, such as prices and timestamps.
* Preference for fixed positions and fixed length fields, supporting direct access to data and avoiding the need for management of heaps of variable-length elements which must be sequentially processed.

## Message schema

This standard describes how fields are encoded and the general structure of messages. The content of a message type is specified by a message schema. A message schema tells which fields belong to a message and their location within a message. Additionally, the metadata describes valid value ranges and information that need not be sent on the wire, such as constant values.

Message schemas may be based on standard FIX message specifications, or may be customized as needed by agreement between counterparties.

## Glossary

**Data type** – A field type with its associated encoding attributes, including backing primitive types and valid values or range. Some types have additional attributes, e.g. epoch of a date.

**Encoding**—a message format for interchange. The term is commonly used to mean the conversion of one data format to another, such as text to binary. However, Simple Binary Encoding strives to use native binary data types in order to make conversion unnecessary, or at least trivial. Encoding also refers to the act of formatting a message, as opposed to decoding.

**Message schema** – metadata that specifies messages and their data types and identifiers. Message schemas may be disseminated out of band. For Simple Binary Encoding, message schemas are expressed as an XML document that conforms to an XML schema that is published as part of this standard.

**Message template** – metadata that specifies the fields that belong to one particular message type. A message template is contained by a message schema.

**Session protocol**—a protocol concerned with the reliable delivery of messages over a transport. FIX protocol makes a distinction between session protocol and the encoding of a message payload, as described by this document. See the specifications section of FIX protocol web site for supported protocols. The original FIX session protocol is known as FIXT.

**XML schema**—defines the elements and attributes that may appear in an XML document. The SBE message schema is defined in W3C (XSD) schema language since it is the most widely adopted format for XML schemas.

## Documentation

This document explains:

* The binary type system for field encoding
* Message structure, including field arrangement, repeating groups, and relationship to a message header that may be provided by a session protocol.
* The Simple Binary Encoding message schema.

### Specification terms

These key words in this document are to be interpreted as described [in Internet Engineering Task Force RFC 2119](http://www.apps.ietf.org/rfc/rfc2119.html). These terms indicate an absolute requirement for implementations of the standard: “**must**”, or “**required**”.

This term indicates an absolute prohibition: “**must not**”.

These terms indicate that a feature is allowed by the standard by not required: “**may**”, “**optional**”. An implementation that does not provide an optional feature must be prepared to interoperate with one that does.

These terms give guidance, recommendation or best practices: “**should**” or “**recommended**”. A recommended choice among alternatives is described as “**preferred**”.

These terms give guidance that a practice is not recommended: “**should not**” or “**not recommended**”.

### Document format

In this document, these formats are used for technical specifications and data examples.

This is a sample encoding specification

<type name="short" primitiveType="int16" semanticType="int" />

This is sample data as it would be transmitted on the wire

10270000

## References

For FIX semantics, see the current FIX message specification, which is currently [FIX Version 5.0 Service Pack 2](http://www.fixprotocol.org/specifications/).

XML 1.1 schema standards are located here [W3C XML Schema](http://www.w3.org/XML/Schema.html#dev)

# Field Encoding

## Field aspects

A field is a unit of data contained by a FIX message. Every field has the following aspects: semantic data type, encoding, and metadata. They will be specified in more detail in the sections are data type encoding and message schema but are introduced here as an overview.

### Semantic data type

The FIX semantic data type of a field tells a data domain in a broad sense, for example, whether it is numeric or character data, or whether it represents a time or price. Simple Binary Encoding represents all of the semantic data types that FIX protocol has defined across all encodings. In message specifications, FIX data type is declared with attribute semanticType. See the section 2.2 below for a listing of those FIX types.

### Encoding

Encoding tells how a field of a specific data type is encoded on the wire. An encoding maps a FIX data type to either a simple, primitive data type, such as a 32 bit signed integer, or to a composite type. A composite type is composed of two or more simple types. For example, the FIX data type Price is encoded as a decimal, a composite type containing a mantissa and an exponent. Note that many fields may share a data type and an encoding. The sections that follow explain the valid encodings for each data type.

### Metadata

Field metadata, part of a message schema, describes a field to application developers. Elements of field metadata are:

* Field ID, also known as FIX tag, is a unique identifier of a field for semantic purposes. For example, tag 55 identifies the Symbol field of an order.
* Field name, as it is known in FIX specifications
* The FIX semantic data type and encoding type that it maps to
* Valid values or data range accepted
* Documentation

Metadata is normally *not* sent on the wire with Simple Binary Encoding messages. It is necessary to possess the message schema that was used to encode a message in order to decode it. In other words, Simple Binary Encoding messages are not self-describing. Rather, message schemas are typically exchanged out-of-band between counterparties.

See section 4 below for a detailed message schema specification.

### Field presence

By default, fields are assumed to be required in a message. However, fields may be specified as optional. To indicate that a value is not set, a special null indicator value is sent on the wire. The null value varies according to data type and encoding. Global defaults for null value may be overridden in a message schema by explicitly specifying the value that indicates nullness.

Alternatively, fields may be specified as constant. In that case, the data is not sent on the wire, but may be treated as constants by applications.

## FIX data type summary

FIX semantic types are mapped to binary field encodings as follows. See sections below for more detail about each type.

Schema attributes may restrict the range of valid values for a field. See Common schema attributes below.

| FIX semantic type | Binary type | Section | Description |
| --- | --- | --- | --- |
| int | Integer encoding | 2.4 | An integer number |
| Length | Integer encoding | 2.4 | Field length in octets. Value must be non-negative. |
| TagNum | Integer encoding | 2.4 | A field's tag number. Value must be positive. |
| SeqNum | Integer encoding | 2.4 | A field representing a message sequence number. Value must be positive |
| NumInGroup | Group dimension encoding | 3.4.8 | A counter representing the number of entries in a repeating group. Value must be positive. |
| DayOfMonth | Integer encoding | 2.4 | A field representing a day during a particular month (values 1 to 31). |
| Qty | Decimal encoding | 2.5 | A number representing quantity of a security, such as shares. The encoding may constrain values to integers, if desired. |
| float | Float encoding | 2.6 | A fractional number |
| Price | Decimal encoding | 2.5 | A decimal number representing a price |
| PriceOffset | Decimal encoding | 2.5 | A decimal number representing a price offset, which can be mathematically added to a Price. |
| Amt | Decimal encoding | 2.5 | A field typically representing a Price times a Qty. |
| Percentage | Decimal encoding | 2.5 | A field representing a percentage (e.g. 0.05 represents 5% and 0.9525 represents 95.25%). |
| char | Character | 2.6.1 | Single ASCII character value. Can include any alphanumeric character or punctuation. All char fields are case sensitive (i.e. m != M). |
| String | Fixed size character array | 2.6.2 | A fixed-length character array of ASCII encoding |
| String | Variable length string | 2.6.3 | Alpha-numeric free format strings can include any character or punctuation. All String fields are case sensitive (i.e. morstatt != Morstatt). ASCII encoding. |
| String—EncodedText | String encoding | 2.6.3 | Non-ASCII string. The character encoding may be specified by a schema attribute. |
| data | String encoding | 2.6.3 | Variable-length data. Must be paired with a Length field. |
| XMLData | String encoding | 2.6.3 | Variable-length XML. Must be paired with a Length field. |
| Country | Fixed length character array; size = 2 or a subset of values may use Enumeration encoding | 2.6.2 | ISO 3166 Country code |
| Currency | Fixed length character array; size = 3 or a subset of values may use Enumeration encoding | 2.6.2 | ISO 4217 Currency code (3 character) |
| Exchange | Fixed length character array; size = 4 or a subset of values may use Enumeration encoding | 2.6.2 | ISO 10383 Market Identifier Code (MIC) |
| Language | Fixed length character array; size = 2 or a subset of values may use Enumeration encoding | 2.6.2 | National language - uses ISO 639-1 standard |
| Implicit enumeration—char or int | Enumeration encoding | 2.11 | A single choice of alternative values |
| Boolean | Boolean encoding | 2.11.5 | Values true or false |
| MultipleCharValue | Multi-value choice encoding | 2.12 | Multiple choice of a set of values |
| MultipleStringValue | Multi-value choice encoding**.** String choices must be mapped to int values. | 2.12 | Multiple choice of a set of values |
| MonthYear | MonthYear encoding | 2.7 | A flexible date format that must include month and year at least, but may also include day or week. |
| UTCTimestamp | Date and time encoding | 2.8 | Time/date combination represented in UTC (Universal Time Coordinated, also known as "GMT") |
| UTCTimeOnly | Date and time encoding | 2.8 | Time-only represented in UTC (Universal Time Coordinated, also known as "GMT") |
| UTCDateOnly | Date and time encoding | 2.8 | Date represented in UTC (Universal Time Coordinated, also known as "GMT") |
| LocalMktDate | Local date encoding | 2.8 | Local date(as oppose to UTC) |
| TZTimeOnly | TZTimeOnly | 2.10.3 | Time of day |
| TZTimestamp | TZTimestamp | 2.10.1 | Tme/date combination representing local time with an offset to UTC to allow identification of local time and timezone offset of that time. The representation is based on ISO 8601 |

## Common field schema attributes

Schema attributes alter the range of valid values for a field. Attributes are optional unless specified otherwise.

| Schema attribute | Description |
| --- | --- |
| presence=required | The field must always be set. This is the default presence. Mutually exclusive with nullValue. |
| presence=constant | The field has a constant value that need not be transmitted on the wire. Mutually exclusive with value attributes. |
| presence=optional | The field has a constant value that need not be transmitted on the wire. |
| nullValue | A special value that indicates that an optional value is not set. See encodings below for default nullValue for each type. Mutually exclusive with presence=required and constant. |
| minValue | The lowest valid value of a range. Applies to scalar data types, but not to String or data types. |
| maxValue | The highest valid value of a range (inclusive unless specified otherwise). Applies to scalar data types, but not to String or data types. |
| semanticType | Tells the FIX semantic type of a field or encoding. It may be specified on either a field or its encoding. |

### Inherited attributes

The attributes listed above apply to a field element or its encoding (wire format). Any attributes specified on an encoding are inherited by fields that use that encoding.

## Integer encoding

Integer encodings should be used for cardinal or ordinal number fields.

### Primitive type encodings

Numeric data types may be specified by range and signed or unsigned attribute. Integer types are intended to convey common platform primitive data types as they reside in memory. An integer type should be selected to hold the maximum range of values that a field is expected to hold.

| Primitive type | Description | Length (octets) |
| --- | --- | --- |
| int8 | Signed byte | 1 |
| uint8 | Unsigned byte / single-byte character | 1 |
| int16 | 16-bit signed integer | 2 |
| uint16 | 16-bit unsigned integer | 2 |
| int32 | 32-bit signed integer | 4 |
| uint32 | 32-bit unsigned integer | 4 |
| int64 | 64-bit signed integer | 8 |
| uint64 | 64-bit unsigned integer | 8 |

### Range attributes for integer fields

The default data ranges and null indicator are listed below for each integer encoding.

A message schema may optionally specify a more restricted range of valid values for a field.

For optional fields, a special null value is used to indicate that a field value is not set. The default null indicator may also be overridden by a message schema.

Required and optional fields of the same primitive type have the same data range. The null value must not be set for a required field.

| Type:  Schema attribute | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| minValue | –127 | 0 | –32767 | 0 | –231 + 1 | 0 | –263 + 1 | 0 |
| maxValue | 127 | 254 | 32767 | 65534 | 231 – 1 | 232 – 2 | 263 – 1 | 264 – 2 |
| nullValue | –128 | 255 | –32768 | 65535 | –231 | 232 – 1 | –263 | 264 – 1 |

### Byte order

The byte order of integer fields, and for derived types that use integer components, is specified globally in a message schema. Little-Endian order is the default encoding, meaning that the least significant byte is serialized first on the wire.

See section 4.2.1 for specification of message schema attributes, including byteOrder. Message schema designers should specify the byte order most appropriate to their system architecture and that of their counterparties.

### Integer encoding specifications

By nature, integers map to simple encodings. These are valid encoding specifications for each of the integer primitive types.

<type name="int8" primitiveType="int8" />

<type name="int16" primitiveType="int16" />

<type name="int32" primitiveType="int32" />

<type name="int64" primitiveType="int64" />

<type name="uint8" primitiveType="uint8" />

<type name="uint16" primitiveType="uint16" />

<type name="uint32" primitiveType="uint32" />

<type name="uint64" primitiveType="uint64" />

### Examples of integer fields

Examples show example schemas and encoded bytes on the wire as hexadecimal digits in Little-Endian byte order.

Example integer field specification

<field type="uint32" name="ListSeqNo" id="67" semanticType="int"

description="Order number within the list" />

Value on the wire - uint32 value decimal 10,000, hexadecimal 2710.

10270000

Optional field with a valid range 0–6

<type name="range06" primitiveType=”uint8” maxValue=”6” presence=”optional” nullValue=”255” />

<field type="range06" name="MaxPriceLevels" id="1090" semanticType="int"/>

Wire format of uint8 value decimal 3.

03

Sequence number field with integer encoding

<field type=*"*uint64*"* name=*"MsgSeqNum"* id=*"34"* semanticType="SeqNum" />

Wire format of uint64 value decimal 100,000,000,000, hexadecimal 174876E800.

00e8764817000000

Wire format of uint16 value decimal 10000, hexadecimal 2710.

1027

Wire format of uint32 null value 232 – 1

ffffffff

## Decimal encoding

Decimal encodings should be used for prices and related monetary data types like PriceOffset and Amt.

FIX specifies Qty as a float type to support fractional quantities. However, decimal encoding may be constrained to integer values if that is appropriate to the application or market.

### Composite encodings

Prices are encoded as a scaled decimal, consisting of a signed integer mantissa and signed exponent. For example, a mantissa of 123456 and exponent of -4 represents the decimal number 12.3456.

#### Mantissa

Mantissa represents the significant digits of a decimal number. Mantissa is a commonly used term in computing, but it is properly known in mathematics as significand or coefficient.

#### Exponent

Exponent represents scale of a decimal number as a power of 10.

#### Floating point and fixed point encodings

A floating-point decimal transmits the exponent on the wire while a fixed-point decimal specifies a fixed exponent in a message schema. A constant negative exponent specifies a number of assumed decimal places to the right of the decimal point.

Implementations should support both 32 bit and 64 bit mantissa. The usage depends on the data range that must be represented for a particular application. It is expected that an 8 bit exponent should be sufficient for all FIX uses.

| Encoding type | Description | Backing primitives | Length (octets) |
| --- | --- | --- | --- |
| decimal | Floating-point decimal | Composite:  int64 mantissa  int8 exponent | 9 |
| decimal64 | Fixed-point decimal | int64 mantissa  constant exponent | 8 |
| decimal32 | Fixed-point decimal | int32 mantissa  constant exponent | 4 |

Optionally, implementations may support any other signed integer types for mantissa and exponent.

### Range attributes for decimal fields

The default data ranges and null indicator are listed below for each decimal encoding.

A message schema may optionally specify a more restricted range of valid values for a field. For optional fields, a special mantissa value is used to indicate that a field value is null.

| Type:  Schema attribute | decimal | decimal32 | decimal64 |
| --- | --- | --- | --- |
| exponent range | Valid range –128 to 127 | Valid range –128 to 127 | Valid range –128 to 127 |
| minValue | –(263 +1)\*10127–10–128 | –(232 +1)\*10–127 | –(263 +1)\*10127–10–128 |
| maxValue | 264 (263–1)\*10127 | 232 (231–1)\*10127 | 264 (263 –1)\*10127 |
| nullValue | mantissa=–263 | mantissa =–231 | mantissa =–263 |

### Encoding specifications for decimal types

Decimal encodings are composite types, consisting of two subfields, mantissa and exponent. The exponent may either be serialized on the wire or may be set to constant. A constant exponent is a way to specify an assumed number of decimal places.

Decimal encoding specifications that an implementation must support

<composite name="decimal" >

<type name="mantissa" primitiveType="int64" />

<type name="exponent" primitiveType="int8" />

</composite>

<composite name="decimal32" >

<type name="mantissa" primitiveType="int32" />

<type name="exponent" primitiveType="int8" presence="constant">-2</type>

</composite>

<composite name="decimal64">

<type name="mantissa" primitiveType="int64" />

<type name="exponent" primitiveType="int8" presence="constant">-2</type>

</composite>

### Composite encoding packing

Composite encodings must not have padding *within* a field. In other words, subfields must be packed at an octet level. This rule applies when both mantissa and exponent are sent on the wire for a decimal.

### Examples of decimal fields

Examples show encoded bytes on the wire as hexadecimal digits, little-endian.

FIX Qty data type is a float type, but a decimal may be constrained to integer values by setting exponent to zero.

<composite name="intQty32" semanticType="Qty">

<type name="mantissa" primitiveType="int32" />

<type name="exponent" primitiveType="int8" presence="constant">0</type>

</composite>

Field inherits semanticType from encoding

<field type="intQty32" name="OrderQty" id="38"

description="Total number of shares" />

Wire format of decimal 123.45 with 2 significant decimal places.

3930000000000000fe

Wire format of decimal64 123.45 with 2 significant decimal places. Schema attribute exponent = –2

3930000000000000

Wire format of decimal32 123.45 with 2 significant decimal places. Schema attribute exponent = –2

39300000

## Float encoding

Binary floating point encodings are compatible with IEEE Standard for Floating-Point Arithmetic (IEEE 754). They should be used for floating point numeric fields that do not represent prices or monetary amounts. Examples include interest rates, volatility and dimensionless quantities such as ratios. On the other hand, decimal prices should be encoded as decimals; see section 2.5 above.

### Primitive types

Both single and double precision encodings are supported as primitive data types. See the IEEE 754 standard for ranges and details of the encodings.

| Primitive type | Description | Length (octets) |
| --- | --- | --- |
| float | Single precision floating point | 4 |
| double | Double precision floating point | 8 |

### Null values

For both float and double precision encodings, null value of an optional field is represented by the Not-a-Number format (NaN) of the standard encoding. Technically, it indicated by the so-called quiet NaN.

### Byte order

Like integer encodings, floating point encodings follow the byte order specified by message schema. See section 4.2.1 for specification of message schema attributes, including byteOrder.

### Float encoding specifications

These are valid encoding specifications for each of the floating point primitive types.

<type name="float" primitiveType="float" />

<type name="double" primitiveType="double" />

### Examples of floating point fields

Examples show encoded bytes on the wire as hexadecimal digits, little-endian.

A single precision ratio

<type name="ratio" primitiveType="float" />

<field type="ratio" name="CurrencyRatio" id="1382"

semanticType=”float”/>

Wire format of float 255.678

91ad7f43

Wire format of double 255.678

04560e2db2f56f40

## String encodings

Character data may either be of fixed size or variable size. In Simple Binary Encoding, fixed-length fields are recommended in order to support direct access to data. Variable-length encoding should be reserved for raw data that cannot be constrained to a specific size.

### Character

Character fields hold a single-character. They are most commonly used for field with character code enumerations. See section 2.11 below for discussion of enum fields.

| FIX data type | Description | Backing primitive | Length (octet) |
| --- | --- | --- | --- |
| char | A single ASCII character | char | 1 |

#### Range attributes for char fields

Valid values of a char field are printable characters of the US-ASCII character set (codes 20 to 7E hex.) The implicit nullValue is the NUL control character (code 0).

| Type:  Schema attribute | char |
| --- | --- |
| minValue | hex 20 |
| maxValue | hex 7e |
| nullValue | 0 |

#### Encoding of char type

This is the standard encoding for char type.

<type name="char" primitiveType="char" semanticType="char" />

Wire format of char encoding of “A” (ASCII value 65, hexadecimal 41)

41

### Fixed-length character array

Character arrays are allocated a fixed space in a message, supporting direct access to fields. A fixed size character array is distinguished from a variable length string by the presence of a length schema attribute or a constant attribute.

| FIX data type | Description | Backing primitives | Length (octets) | Required schema attribute |
| --- | --- | --- | --- | --- |
| String | ASCII character array | Array of char of specified length, delimited by NUL character if a string is shorter than the length specified for a field. | Specified by length attribute | length (except may be inferred from a constant value, if present). |

A length attribute set to zero indicates variable length. See section 2.7.3 below for variable-length data encoding.

#### Encoding specifications for fixed-length character array

A fixed-length character array encoding must specify primitiveType="char" and a length attribute is required. Each element of the array is an ASCII character.

Data range attributes minValue and maxValue do not apply to String data type.

#### Examples of fixed-length character arrays

A typical string encoding specification

<type name="string6" primitiveType="char" semanticType="String"

length="6" />

<field type=*"string6"* name=*"Symbol"* id=*"55"* />

Wire format of a character array in character and hexadecimal formats

M S F T

4d5346540000

A character array constant specification

<type name="EurexMarketID" semanticType="Exchange" primitiveType="char" length="4" description="MIC code" presence="constant">XEUR</type>

<field type=*"EurexMarketID"* name=*"MarketID"* id=*"1301"* />

### Variable-length data encoding

Variable-length data is used for variable length ASCII strings or embedded non-ASCII character data (like EncodedText field), XML (semantic data type XmlData), or non-character data (such as RawData). A separate length field coveys the size of the field.

On the wire, length immediately precedes the data.

The length subfield may be set to zero for an empty string. In that case, no space is reserved for the data.

| FIX data type | Description | Backing primitives | Length (octets) |
| --- | --- | --- | --- |
| Length | The length of variable data in octets | primitiveType=uint8 or uint16 | 1 or 2 |
| data | Raw data | Array of char of size specified in associated Length field. The data field itself should be specified as variable length.  primitiveType="uint8" or "char"  length="0" indicates variable length | variable |

The primitive type of the data element should be uint8 for non-ASCII data or char to constrain it to ASCII character set. Other data representation should be specified as uint8. Usage of variable-length arrays of other primitive types is currently undefined.

### Range attributes for variable-length data fields

Key

Y=required

N=not required

X=not allowed

| Schema attribute | length  uint8 | length  uint16 | data |
| --- | --- | --- | --- |
| minValue | 0 | 0 | X |
| maxValue | 254 | 65534 | X |
| nullValue | 255 | 65535 | X |

If the Length field has minValue and maxValue attributes, it specifies the minimum and maximum *length* of the variable-length data. Data range attributes minValue and maxValue do not apply to a data field.

If a field is required, both the Length and data fields must be set to a “required” attribute.

The encoding attribute tells which variable-sized encoding is used if the raw data field represents encoded text. UTF-8 is the default encoding.

### Encoding specifications for variable-length data

Variable length data is encoded as composite type, consisting of a length sub field and data subfield. The length attribute of the varData element is set to zero in the XML message schema as special value to indicate that the character data is of variable length.

To map an SBE data field specification to traditional FIX, the field ID of a data field is used. Its associated length is implicitly contained by the composite type rather than specified as a separate field.

Encoding specification for variable length data up to 65535 octets

<composite name="DATA" description="Variable-length data">

<type name="length" primitiveType="uint16"

semanticType="Length"/>

<type name="varData" length="0" primitiveType="uint8"

semanticType="data"/>

</composite>

<data name="RawData" id="96" type="DATA"/>

### Example of a data field

Example shows encoded bytes on the wire.

Wire format of data in character and hexadecimal formats, preceded by uint16 length of 4 octets in little-endian byte order

M S F T

04004d53465

## MonthYear encoding

MonthYear encoding contains four subfields representing respectively year, month, and optionally day or week. A field of this type is not constrained to one date format. One message may contain only year and month while another contains year, month and day in the same field, for example.

Values are distinguished by position in the field. Year and month must always be populated for a non-null field. Day and week are set to special value indicating null if not present. If Year is set to the null value, then the entire field is considered null.

| Subfield | Primitive type | Length (octets) | Null value |
| --- | --- | --- | --- |
| Year | uint16 | 2 | 65535 |
| Month (1-12) | uint8 | 1 | — |
| Day of the month(1-31) optional | uint8 | 1 | 255 |
| Week of the month (1-5) optional | uint8 | 1 | 255 |

### Composite encoding packing

Composite encodings must not have padding *within* a field. The four subfields of MonthYear must be packed at an octet level.

### Encoding specifications for MonthYear

MonthYear data type is based on a composite encoding that carries its required and optional elements.

The standard encoding specification for MonthYear

<composite name="monthYear" semanticType="MonthYear">

<type name="year" primitiveType="uint16" presence="optional" nullValue="65536" />

<type name="month" primitiveType="uint8" minValue="1" maxValue="12" />

<type name="day" primitiveType="uint8" minValue="1" maxValue="31" presence="optional" nullValue="255" />

<type name="week" description="week of month" primitiveType="uint8" minValue="1" maxValue="5" presence="optional" nullValue="255" />

</composite>

Example MonthYear field specification

<field type=*"monthYear"* name=*"MaturityMonthYear"* id=*"200"* />

Wire format of MonthYear 2014 June week 3 as hexadecimal

de0706ff03

## Date and time encoding

Dates and times represent Coordinated Universal Time (UTC). This is the preferred date/time format, except where regulations require local time with time zone to be reported (see time zone encoding below).

### Epoch

Each time type has an epoch, or start of a time period to count values. For timestamp and date, the standard epoch is the UNIX epoch, midnight January 1, 1970 UTC.

A time-only value may be thought of as a time with an epoch of midnight of the current day.

Like current time, the epoch is also referenced as UTC.

### Time unit

Time unit tells the precision at which times can be collected. Binary timestamps default to nanosecond precision. Precision may be specified in a message schema to inform consumers of actual clock precision.

| FIX data type | Description | Backing primitives | Length (octets) | Schema attributes |
| --- | --- | --- | --- | --- |
| UTCTimestamp | UTC date/time  Default: nanoseconds since Unix epoch Range Jan. 1, 1970 - July 21, 2554 | uint64 | 8 | epoch=”unix” (default) |
|  |  |  |  | timeUnit = second | millisecond | microsecond | nanosecond (default) |
| UTCTimeOnly | UTC time of day only  Default: nanoseconds since midnight today | uint64 | 8 | timeUnit = second | millisecond | microsecond | nanosecond (default) |
| UTCDateOnly | UTC calendar date  Default: days since Unix epoch. Range: Jan. 1, 1970 - June 7, 2149 | uint16 | 2 | epoch=”unix” (default) |

### Examples of date/time fields

**timestamp** 14:17:22 Friday, October 4, 2024 (20,000 days and 14 hours since epoch) with default schema attributes

<type name=*"timestamp"* primitiveType=*"uint64"* timeUnit=*"nanosecond"* semanticType=*"UTCTimestamp"* />

Wire format of UTCTimestamp

007420bf8838fb17

**time** 10:24:39.123456000 (37,479 seconds and 123456000 nanoseconds since midnight UTC) with default schema attributes

*<type name="time" primitiveType="uint64" timeUnit="nanosecond"*  *semanticType="UTCTimeOnly" />*

Wire format of UTCTimeOnly

42dc561600000000

**date** Friday, October 4, 2024 (20,000 days since UNIX epoch) with default schema attributes

<type name=*"date"* primitiveType=*"uint16"*

semanticType=*"UTCDateOnly"* />

Wire format of UTCDateOnly

204e

## Local date encoding

Local date is encoded the same as UTCDateOnly, but it represents local time at the market instead of UTC time.

| FIX data type | Description | Backing primitives | Length (octets) | Schema attributes |
| --- | --- | --- | --- | --- |
| LocalMktDate | Local calendar date  Default: days since Unix epoch. Range: Jan. 1, 1970 - June 7, 2149 local time | uint16 | 2 | epoch=”unix” (default)  Represents Jan. 1, 1970 local time |

The standard encoding specification for LocalMktDate

<type name="localMktDate" primitiveType="uint16" semanticType="LocalMktDate" />

## Local time encoding

Time with time zone encoding should only be used when required by market regulations. Otherwise, use UTC time encoding (see above).

Time zone is represented as an offset from UTC in the ISO 8601 format ±hhmm.

### TZTimestamp encoding

A binary UTCTimestamp followed by a number representing the time zone indicator as defined in ISO 8601.

| FIX data type | Description | Backing primitives | Length (octets) | Schema attributes |
| --- | --- | --- | --- | --- |
| TZTimestamp | date/time with timezone | uint64  Default: nanoseconds since Unix epoch Range Jan. 1, 1970 - July 21, 2554 | 8 | epoch=”unix” (default)  Represents Jan. 1, 1970 local time |
|  |  |  |  | timeUnit = second | millisecond | microsecond | nanosecond (default) |
|  | Time zone hour offset | int8 | 1 | None |
|  | Time zone minute offset | uint8 | 1 | None |

### Composite encoding packing

Composite encodings must not have padding *within* a field. The subfields of TZTimestamp must be packed at an octet level.

Standard TZTimestamp encoding specification

<composite name="tzTimestamp" semanticType="TZTimestamp">

<type name="time" primitiveType="uint64" timeUnit="nanosecond" />

<!-- Sign of timezone offset is on hour subfield -->

<type name="timezoneHour" primitiveType="int8"

minValue="-12" maxValue="14" />

<type name="timezoneMinute" primitiveType="uint8" maxValue="59" />

</composite>

Wire format of TZTimestamp 8:30 17 September 2013 with Chicago time zone offset (-6:00)

0050d489fea22413fa00

### TZTimeOnly encoding

A binary UTCTimeOnly followed by a number representing the time zone indicator as defined in ISO 8601.

The time zone hour offset tells the number of hours different to UTC time. The time zone minute tells the number of minutes different to UTC. The sign telling ahead or behind UTC is on the hour subfield.

| FIX data type | Description | Backing primitives | Length (octets) | Schema attributes |
| --- | --- | --- | --- | --- |
| TZTimeOnly | Time of day only with time zone | uint64  Default: nanoseconds since midnight today, local time | 8 | timeUnit = second | millisecond | microsecond | nanosecond (default) |
|  | Time zone hour offset | int8 | 1 | None |
|  | Time zone minute offset | uint8 | 1 | None |

### Composite encoding packing

Composite encodings must not have padding *within* a field. The subfields of TZTimeOnly must be packed at an octet level.

Standard TZTimeOnly encoding specification

<composite name="tzTimeOnly" semanticType="TZTimeOnly">

<type name="time" primitiveType="uint64" timeUnit="nanosecond" />

<!-- Sign of timezone offset is on hour subfield -->

<type name="timezoneHour" primitiveType="int8"

minValue="-12" maxValue="14" />

<type name="timezoneMinute" primitiveType="uint8" minValue="0" maxValue="59" />

</composite>

Wire format of TZTimeOnly 8:30 with Chicago time zone offset (-6:00)

006c5ebe76000000fa00

## Enumeration encoding

An enumeration conveys a single choice of mutually exclusive valid values.

### Primitive type encodings

An unsigned integer or character primitive type is selected to contain the number of choices.

| Primitive type | Description | Length (octets) | Maximum number of choices |
| --- | --- | --- | --- |
| char | character | 1 | 95 |
| uint8 | 8-bit unsigned integer | 1 | 255 |

### Value encoding

If a field is of FIX data type char, then its valid values are restricted to ASCII printable characters. See section 2.6.1 above.

If the field is of FIX data type int, then a primitive integer data type should be selected that can contain the number of choices. For most cases, an 8 bit integer will be sufficient, allowing 255 possible values.

### Enumeration of values

In a message schema, the choices are specified a <validValue> members of an <enum>. An <enum> specification must contain at least one <validValue>.

The name and value of a validValue element must be unique within an enumeration.

### Enumeration examples

These examples use a char field is for enumerated code values.

Example enum lists acceptable values and gives the underlying encoding, which in this case is char.

<enum *SideEnum*="Side" encodingType="char">

<validValue name="Buy">1</validValue>

<validValue name="Sell">2</validValue>

<validValue name="SellShort">5</validValue>

<validValue name="SellShortExempt">6</validValue>

<!-- not all FIX values shown -->

</enum>

Side field specification references the enumeration

<field type=*"Side"* name=*"SideEnum"* id=*"54"* />

Wire format of Side “Buy” code as hexadecimal

01

### Constant field of an enumeration value

A constant field may be specified as a value of an enumeration. The attribute valueRef is a cross-reference to validValue entry by symbolic name.

Example of a char field using a constant enum value

<enum name=*"PartyIDSourceEnum"* primitiveType="char">

<validValue name="BIC">B</validValue>

<validValue name="GeneralIdentifier">C</validValue>

<validValue name="Proprietary">D</validValue>

</enum>

<field type=*"PartyIDSourceEnum"* name=*"PartyIDSource"* id=*"447"* description=”Party ID source is fixed” presence="constant"

valueRef=”GeneralIdentifier” />

### Boolean encoding

A Boolean field is a special enumeration with predefined valid values: true and false. Like a standard enumeration, an optional Boolean field may have nullValue that indicates that the field is null (or not applicable).

Standard encoding specifications for required and optional Boolean fields

<type name="uint8" primitiveType="uint8" />

<enum name="boolean" encodingType="uint8" semanticType="Boolean">

<validValue name="false">0</validValue>

<validValue name="true">1</validValue>

</enum>

<enum name="optionalBoolean" encodingType="uint8" presence="optional" nullValue="255" semanticType="Boolean">

<validValue name="false">0</validValue>

<validValue name="true">1</validValue>

</enum>

Example optional Boolean field

<field type=*"*optionalBoolean*"* name=*"SolicitedFlag"* id=*"377"* />

Wire format of true value as hexadecimal

01

Wire format of false value as hexadecimal

00

Wire format of null Boolean (or N/A) value as hexadecimal

ff

## Multi-value choice encoding

A multi-value field conveys a choice of zero or more non-exclusive valid values.

### Primitive type encodings

The binary encoding uses a bit set to represent up to 64 possible choices. The smallest unsigned primitive type should be selected that can contain the number of valid choices.

| Primitive type | Description | Length (octets) | Maximum number of choices |
| --- | --- | --- | --- |
| uint8 | 8-bit unsigned integer | 1 | 8 |
| uint16 | 16-bit unsigned integer | 2 | 16 |
| uint32 | 32-bit unsigned integer | 4 | 32 |
| uint64 | 64-bit unsigned integer | 8 | 64 |

### Value encoding

Each choice is assigned a bit of the primitive integer encoding, starting with the least significant bit. For each choice the value is selected or not, depending on whether it corresponding bit is set or cleared.

Any remaining unassigned bits in an octet should be cleared.

There is no explicit null value for multi-value choice encoding other than to set all bits off when no choices are selected.

### Enumeration of values

In a message schema, the choices are specified as <choice> members of an <set> element. Choices are assigned values as an ordinal of bits in the bit set. The first Choice “0” is assigned the least significant bit; choice “1” is the second bit, and so forth.

The name and value (bit position) must be unique for element of a set.

### Multi-value example

Example of a multi-value choice (was MultipleCharValue in tag-value encoding)

<set name="FinancialStatusEnum” encodingType="uint8"">

< choice name="Bankrupt">0</choice>

< choice name="Pending delisting">1</choice>

< choice name= Restricted">2</choice>

</set>

<field type=*"*FinancialStatus*"* name=*"*FinancialStatusEnum*"*

id=*"291"* semanticType=”MultipleCharValue”/>

Wire format of choices “Bankrupt” + “Pending delisting” (first and second bits set)

03

## Field value validation

These validations apply to message field values.

If a value violation is detected on a received message, the message should be rejected back to the counterparty in a way appropriate to the session protocol.

| Error condition | Error description |
| --- | --- |
| Field value less than minValue | The encoded value falls below the specified valid range. |
| Field value greater than maxValue | The encoded value exceeds the specified valid range. |
| Null value set for required field | The null value of a data type is invalid for a required field. |
| String contains invalid characters | A String contains non-ASCII printable characters. |
| Required subfields not populated in MonthYear | Year and month must be populated with non-null values, and the month must be in the range 1-12. |
| UTCTimeOnly exceeds day range | The value must not exceed the number of time units in a day, e.g. greater than 86400 seconds. |
| TZTimestamp and TZTimeOnly has missing or invalid time zone | The time zone hour and minute offset subfields must correspond to an actual time zone recognized by international standards. |
| Value must match valid value of an enumeration field | A value is invalid if it does not match one of the explicitly listed valid values. |

# Message Structure

## Session Protocol and Message Frame

FIX protocol makes a distinction between message encoding, as described by this document, and session protocol, which is concerned with reliable delivery of messages over a transport. Therefore, headers will be divided between sections for the session protocol and Simple Binary Encoding.

The layout of a session protocol header is specified by that protocol’s own documentation. It typically includes message sequence and the like, but that is not needed for message body decoding.

The items that do relate to decoding messages are:

* Message framing, the elements that delimit messages on a stream
* Message size

For example, in FIXT, the BeginString header field and CheckSum trailer field delimit messages. The BodyLength header field tells the overall size of a message. Other session protocols have other mechanisms for framing and conveying message size.

**Tip**

The encoding of a message header defined by a session protocol may be different than the field encoding specified by SBE.

### Minimal frame header

If there is no requirement for a session protocol to ensure reliable message delivery, then the SBE message may be framed by prefixing it with a message size field. This tells a consumer how many octets to read from a stream. The size represents the entire message, including headers and message body.

Message size for framing should be encoded in SBE as an unsigned integer type.

Recommended minimal frame header encoding

<type name="messageSize" primitiveType="uint16"/>

## SBE Message Header

Within the frame delivered by the session protocol, an SBE message begins with its own header. The purpose of the message header is to tell which message schema was used to encode the message and to give information about the size of the message body to aid in decoding.

The fields of the SBE message header are:

* **Template ID**—identifier of the message template
* **Message schema version**—the version of the message schema in which the message is defined
* **Block length of the message root**—the total space reserved for the root level of the message not counting any repeating groups or variable-length fields.

Block length is specified in a message schema, but it is also serialized on the wire. By default, block length is set to the sum of the sizes of body fields in the message. However, it may be increased to force padding at the end of block. See section 3.3.3.3 below.

### Message header schema

The header fields precede the message body of every message in a fixed position as shown below. Each of these fields must be encoded as an unsigned integer type. Since elements like size are vary with use case, the precise encoding of header fields is configurable in a message schema. The encoding must carry the name “messageHeader”.

The message header is encoded in the same byte order as the message body, as specified in a message schema. See section 4.2.1.

Recommended message header encoding

<composite name="messageHeader" description="Template ID and length of message root">

<type name="blockLength" primitiveType="uint16"/>

<type name="templateId" primitiveType="uint16"/>

<type name="version" primitiveType="uint8"/>

<type name="reserved" primitiveType="uint8"/>

</composite>

| Field | Size | Offset |
| --- | --- | --- |
| Root block length | 2 | 0 |
| Template ID | 2 | 2 |
| Schema Version | 1 | 4 |
| reserved | 1 | 5 |

### Root block length

The total space reserved for the root level of the message not counting any repeating groups or variable-length fields. (Repeating groups have their own block length; see section 3.4 below. Length of a variable-length Data field is given by its corresponding Length field; see section 2.6.3 above.) Block length only represents message body fields; it does not include the length of the message header itself, which is a fixed size.

The block size must be at least the sum of lengths of all fields at the root level of the message, and that is its default value. However, it may be set larger to reserve more space to effect alignment of blocks. This is specified by setting the blockLength attribute in a message schema. See section 4.4.2 below.

### Template ID

The identifier of a message type in a message schema. See section 4.4.2 below for schema attributes of a message.

### Schema version

The version number of the message schema that was used to encode a message. See section 4.2.1 below for schema attributes.

## Message Body

The message body conveys the business information of the message.

### Data only on the wire

In SBE, fields of a message occupy proximate space without delimiters or metadata, such as tags.

### Direct access

Access to data is positional, guided by a message schema that specifies a message type.

Data fields in the message body correspond to message schema fields; they are arranged in the same sequence. The first data field has the type and size specified by the first message schema field, the second data field is described by the second message schema field, and so forth. Since a message decoder follows the field descriptions in the schema for position, it is not necessary to send field tags on the wire.

In the simplest case, a message is flat record with a fixed length. Based on the sequence of field data types, the offset to a given data field is constant for a message type. This offset may be computed in advance, based on a message schema. Decoding a field consists of accessing the data at this fixed location.

### Field position and padding

#### No padding by default

By default, there is no padding between fields. In other words, a field value is packed against values of its preceding and following fields. No consideration is given to byte boundary alignment.

By default, the position of a field in a message is determined by the sum of the sizes of prior fields, as they are defined by the message schema.

<field name="ClOrdId" id="11" type="string14" semanticType="String"/>

<field name="Side" id="54" type="char" semanticType="char"/>

<field name="OrderQty" id="38" type="*intQty32*" semanticType="Qty"/>

<field name="Symbol" id="55" type="string8" semanticType="String"/>

|  |  |  |
| --- | --- | --- |
| Field | Size | Offset |
| ClOrdId | 14 | 0 |
| Side | 1 | 14 |
| OrderQty | 4 | 15 |
| Symbol | 8 | 19 |

#### Field offset specified by message schema

If a message designer wishes to introduce padding or control byte boundary alignment or map to an existing data structure, field offset may optionally be specified in a message schema. Field offset is the number of octets from the start of the message body or group to the first octet of the field. Offset is a zero-based index.

If specified, field offset must be greater than or equal to the sum of the sizes of prior fields. In other words, an offset is invalid if it would cause fields to overlap.

Extra octets specified for padding should never be interpreted as business data. They should be filled with binary zeros.

Example of fields with specified offsets

<field name="ClOrdId" id="11" type="string14" offset="0" semanticType="String"/>

<field name="Side" id="54" type="char" offset="14" semanticType="char"/>

<field name="OrderQty" id="38" type="*intQty32*" offset="16" semanticType="Qty"/>

<field name="Symbol" id="55" type="string8" offset="20" semanticType="String"/>

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Size | Padding preceding field | Offset |
| ClOrdId | 14 | 0 | 0 |
| Side | 1 | 0 | 14 |
| OrderQty | 4 | 1 | 16 |
| Symbol | 8 | 0 | 20 |

#### Padding at end of a message or group

In order to force messages or groups to align on byte boundaries or map to an existing data structure, they may optionally be specified to occupy a certain space with a blockLength attribute in the message schema. The extra space is padded at the end of the message or group. If specified, blockLength must be greater than or equal to the sum of the sizes of all fields in the message or group.

The blockLength attribute applies only to the portion of message that contains fix-length fields; it does not apply to variable-length data elements of a message.

Extra octets specified for padding should be filled with binary zeros.

Example of blockLength specification for 24 octets

<message name="ListOrder" id="2" blockLength="24">

## Repeating Groups

A repeating group is a message structure that contains a variable number of entries. Each entry contains fields specified by a message schema.

The order and data types of the fields are the same for each entry in a group. That is, the entries are homogeneous, and the position of a given field within any entry is fixed.

A message may have no groups or an unlimited number of repeating groups specified in its schema.

### Schema specification of a group

A repeating group is defined in a message schema by adding a <group> element to a message template. An unlimited number of <field> elements may be added to a group, but a group must contain at least one field.

Example repeating group encoding specification

<group name="Parties" id="1012" blockLength="16">

<field name="PartyID" id="448" type="string14" semanticType="String"/>

<field name="PartyIDSource" id="447" type="char" semanticType="char"/>

<field name="PartyRole" id="452" type="uint8" semanticType="int"/>

</group>

### Padding at end of a group entry

By default, the space reserved for an entry is the sum of a group’s field lengths, as defined by a message schema, without regard to byte alignment.

The space reserved for an entry may optionally be increased to effect alignment of entries or to plan for future growth. This is specified by adding the group attribute blockLength to reserve a specified number of octets per entry. If specified, the extra space is padded at the end of each entry and should be set to zeroes by encoders. The blockLength value does not include the group dimensions itself.

### Entry counter

Each group is associated with a required counter field of semantic data type NumInGroup to tell how many entries are contained by a message. The value of the counter is a non-negative integer. See section 3.4.8 below for encoding of that counter.

### Empty group

The space reserved for all entries of a group is the product of the space reserved for each entry times the value of the associated NumInGroup counter. If the counter field is set to zero, then no entries are sent in the message, and no space is reserved for entries. The group dimensions including the zero-value counter is still transmitted, however.

### Multiple repeating groups

A message may contain multiple repeating groups at the same level.

Example of encoding specification with multiple repeating groups

<message name="ExecutionReport" id="8">

<group name="ContraGrp" id=”2012”>

**[ContraGrp group fields]**

</group>

<group name="PreAllocGrp" id=”2026”>

**[PreAllocGrp group fields]**

</group>

</message>

### Nested repeating group specification

Repeating groups may be nested to an arbitrary depth. That is, a <group> in a message schema may contain one or more <group> child elements, each associated with their own counter fields.

The encoding specification of nested repeating groups is in the same format as groups at the root level of a message in a recursive procedure.

Example of nested repeating group specification

<group name="ListOrdGrp" id="2030">

<field name="ClOrdID" id="11" type="string14" semanticType="String"/>

<field name="ListSeqNo" id="67" type="uint32" semanticType="int"/>

<field name="Symbol" id="55" type="string8" semanticType="String"/>

<field name="Side" id="54" type="char"

semanticType="char"/>

<field name="OrderQty" id="38" type="intQty32" semanticType="Qty"/>

<group name="Parties" id=”1012”>

<field name="PartyID" id="448" type="string14" semanticType="String"/>

<field name="PartyRole" id="452" type="int" semanticType="int"/>

</group>

</group>

### Nested repeating group wire format

Nested repeating groups are encoded on the wire by a depth-first walk of the data hierarchy. For example, all inner entries under the first outer entry must be encoded before encoding outer entry 2. (This is the same element order as FIX tag-value encoding.)

On decoding, nested repeating groups do no support direct access to fields. It is necessary to walk all elements in sequence to discover the number of entries in each repeating group.

### Group dimension encoding

Every repeating group must be immediately preceded on the wire by its dimensions. The two dimensions are the count of entries in a repeating group and the space reserved for each entry of the group.

#### Range of group entry count

Implementations should support uint8 and uint16 types for repeating group entry counts. Optionally, implementations may support any other unsigned integer types.

| Primitive type | Description | Length (octets) | Maximum number of entries |
| --- | --- | --- | --- |
| uint8 | 8-bit unsigned integer | 1 | 255 |
| uint16 | 16-bit unsigned integer | 2 | 65535 |

#### Encoding of repeating group dimensions

Conventionally in FIX, a NumInGroup field conveys the number of entries in a repeating group. In SBE, the encoding conveys two dimensions: the number of entries and the length of each entry in number octets. Therefore, the encoding is a composite of those two elements. Block length and entry count subfields must be encoded as an unsigned integer types.

By default, the name of the group dimension encoding is groupSizeEncoding. This name may be overridden by setting the dimensionType attribute of a <group> element.

Recommended encoding of repeating group dimensions

<composite name="groupSizeEncoding">

<type name="blockLength" primitiveType="uint16"/>

<type name="numInGroup" primitiveType="uint16"

semanticType=*"NumInGroup"*/>

</composite>

Wire format of NumInGroup with block length 55 octets by 3 entries

37000300

## Sequence of message body elements

To maximize deterministic field positions, message schemas must be specified with this sequence of message body elements:

1. Fixed-length fields that reside at the root level of the message (that is, not members of repeating groups), including any of the following, in the order specified by the message schema::
   1. Fixed-length scalar fields, such as integers
   2. Fixed-length character arrays
   3. Fixed-length composite types, such as MonthYear
2. Repeating groups, if any.
3. Data fields, including raw data and variable-length strings, if any.

Repeating group entries are recursively organized in the same fashion as the root level: fixed-length fields, then nested repeating groups, and finally, variable-length data fields.

## Message structure validation

Aside from message schema validations (see section 4.7 below), these validations apply to message structure.

If a message structure violation is detected on a received message, the message should be rejected back to the counterparty in a way appropriate to the session protocol.

| Error condition | Error description |
| --- | --- |
| Wrong message size in header | A message size value smaller than the actual message may cause a message to be truncated. |
| Wrong or unknown template ID in header | A mismatch of message schema would likely render a message unintelligible or cause fields to be misinterpreted. |

# Message Schema

## XML namespace

The Simple Binary Encoding XML schema is identified by this URL:

xmlns:sbe=<http://www.fixprotocol.org/ns/simple/1.0>

Conventionally, the URI of the XML schema is aliased by the prefix “sbe”.

## Name convention

All symbolic names in a message schema are restricted to alphanumeric characters plus underscore without spaces. This is the same restriction applied to all names in FIX specifications.

## Root element

The root element of the XML document is <messageSchema>.

### <messageSchema> attributes

The root element provides basic identification of a schema.

The byteOrder attribute controls the byte order of integer encodings within the schema. It is a global setting for all specified messages and their encodings.

| Schema attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| package | Name or category of a schema | string | optional |  |
| version | Version of this schema | non­negative­Integer | default=0 |  |
| semanticVersion | Version of FIX semantics | string | optional | FIX versions, such as “FIX.5.0\_SP2” |
| byteOrder | Byte order of encoding | token | default = littleEndian | littleEndian  bigEndian |
| description | Documentation of the schema | string | optional |  |
| headerType | Name of the encoding type of the message header, which is the same for all messages in a schema. | string | default= messageHeader |  |

### Schema versioning

Changes to a message schema may be tracked by its version attribute. A version of a schema is a snapshot in time. All elements in a given generation of the schema share the same version number. That is, elements are not versioned individually. By convention, the initial version of a schema is version zero, and subsequent changes increment the version number.

The package attribute should remain constant between versions, if it is supplied.

<messageSchema> element

<xs:element name="messageSchema">

<xs:annotation><xs:documentation>

Root of XML document, holds all message templates

and their elements

</xs:documentation></xs:annotation>

<xs:complexType>

[See encoding and message elements below.]

<xs:attribute name="package" type="xs:string" />

<xs:attribute name="version" type="xs:nonNegative­Integer" default=”0” />

<xs:attribute name="semanticVersion" type="xs:string" use="optional" />

<xs:attribute name="description" type="xs:string" use="optional" />

<xs:attribute name="byteOrder" default="littleEndian">

<xs:simpleType>

<xs:restriction base="xs:token">

<xs:enumeration value="bigEndian" />

<xs:enumeration value="littleEndian" />

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<xs:attribute name="headerType" type="xs:string"

default="messageHeader"/>

</xs:complexType>

</xs:element>

Example message schema root element

<?xml version="1.0"?>

<sbe:messageSchema xmlns:sbe="http://www.fixprotocol.org/ns/simple/1.0"

package="orders" version="FIX.5.0" description="My message dictionary" byteOrder="bigEndian">

## Data encodings

### Encoding sets

The <types> element contains one or more sets of data encodings used for messages within the schema.

Within each set, an unbound number of encodings will be listed in this sequence:

1. Element <type> defines a simple encoding
2. Element <composite> defines a composite encoding
3. Element <enum> defines an enumeration
4. Element <set> defines a multi-value choice bitset encoding

### Encoding name

The namespace for encoding names is global across all encodings included in a schema, including simple, composite and enumeration types. That is, the name must be unique among all encoding instances.

All symbolic names should be alphanumeric without spaces.

<types> element

<xs:element name="types" minOccurs="1" maxOccurs="unbounded">

<xs:annotation><xs:documentation>

More than one set of types may be provided, e.g.

built-in and custom.

Names must be unique across all encoding types.

</xs:documentation></xs:annotation>

<xs:complexType>

<xs:sequence>

<xs:element name="type" type="sbe:encodedDataType"

minOccurs="1" maxOccurs="unbounded" />

<xs:element name="composite" type="sbe:compositeDataType"

minOccurs="0" maxOccurs="unbounded" />

<xs:element name="enum" type="sbe:enumType" minOccurs="0" maxOccurs="unbounded" />

<xs:element name="set" type="sbe:setType"

minOccurs="0" maxOccurs="unbounded" />

</xs:sequence>

</xs:complexType>

</xs:element>

#### Importing encodings

A suggested usage is to import common encodings that are used across message schemas as one set while defining custom encodings that are particular to a schema in another set.

Example of XML include usage to import common encoding types

<!-- included XML contains a <types> element -->

<xi:include href="sbe-builtins.xml"/>

### Simple encodings

A simple encoding is backed by either a scalar type or an array of scalars, such as a character array. One or more simple encodings may be defined, each specified by a <type> element.

#### <type> element content

If the element has a value, it is used to indicate a special value of the encoding.

##### Constant value

The element value represents a constant if attribute presence=“constant”. In this case, the value is conditionally required.

#### <type> attributes

| <type> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of encoding | *symbolic­Name\_t* | required | Must be unique among all encoding types in a schema. |
| description | Documentation of the type | string | optional |  |
| presence | Presence of any field encoded with this type | token |  | required  optional  constant |
| nullValue | Special value used to indicate null for an optional field | string | Only valid if presence = optional | The XML string must be convertible to the scalar data type specified by primitiveType. |
| minValue | Lowest acceptable value | string |  |
| maxValue | Highest acceptable value | string |  |
| length | Number of elements of the primitive data type | non­negative­Integer | default = 1 | Value “0” represents variable length.  Arrays and variable-length elements are only defined for char and uint8 elements. |
| primitiveType | The primitive data type that backs the encoding | token | required | char  int8  int16  int32  int64  uint8  uint16  uint32  uint64  float  double |
| semanticType | Represents a FIX data type | token | optional | Same as field semanticType – see below. |
| sinceVersion | Documents the version of a schema in which a type was added | non­negative­Integer | default = 0 |  |

#### FIX data type specification

The attribute semanticType must be specified on either a field or on its corresponding type encoding. It need not be specified in both places, but if it is, the two values must match.

<type> element

<xs:complexType name="encodedDataType" mixed="true">

<xs:annotation><xs:documentation>

Simple data type of a field

</xs:documentation></xs:annotation>

<xs:simpleContent>

<xs:extension base="xs:token">

<xs:attribute name="name" type="sbe:symbolic­Name\_t" use="required" />

<xs:attribute name="description" type="xs:string" use="optional" />

<xs:attribute name="presence" default="required">

<xs:simpleType>

<xs:restriction base="xs:token">

<!-- value must always be populated -->

<xs:enumeration value="required" />

<!-- Value set to nullValue to indicate value not set -->

<xs:enumeration value="optional" />

<!-- Value does not vary -->

<xs:enumeration value="constant" />

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<!-- Special values in string format -->

<!-- Override of default null indicator -->

<xs:attribute name="nullValue" type="xs:string" use="optional" />

<!-- Valid numerical range -->

<xs:attribute name="minValue" type="xs:string" use="optional" />

<xs:attribute name="maxValue" type="xs:string" use="optional" />

<!-- Number of elements of the primitiveType -->

<xs:attribute name="length" type="xs:nonNegativeInteger" default="1" />

<xs:attribute name="primitiveType" use="required">

<xs:simpleType>

<xs:restriction base="xs:token">

<xs:enumeration value="char" />

<xs:enumeration value="int8" />

<xs:enumeration value="int16" />

<xs:enumeration value="int32" />

<xs:enumeration value="int64" />

<xs:enumeration value="uint8" />

<xs:enumeration value="uint16" />

<xs:enumeration value="uint32" />

<xs:enumeration value="uint64" />

<xs:enumeration value="float" />

<xs:enumeration value="double" />

</xs:restriction>

</xs:simpleType>

</xs:attribute>

<xs:attributeGroup ref="sbe:fieldAttributes" />

</xs:extension>

</xs:simpleContent>

</xs:complexType>

Simple type examples

<type name=*"FLOAT"* primitiveType=*"double"*

semanticType=*"float"*/>

<type name="TIMESTAMP" primitiveType="uint64"

semanticType="UTCTimestamp"/>

<type name="GeneralIdentifier" primitiveType="char" description="Identifies class or source

of the PartyID" presence="constant">C</type>

### Composite encodings

Composite encoding types are composed of two or more simple types.

#### <composite> attributes

| <type> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of encoding | symbolic­­Name\_t | required | Must be unique among all encoding types. |
| description | Documentation of the type | string | optional |  |
| semanticType | Represents a FIX data type | token | optional | Same as field semanticType – see below. |
| sinceVersion | Documents the version of a schema in which a type was added | non­negative­Integer | default = 0 |  |

#### Composite type elements

The <type> elements that compose a composite type carry the same XML attributes as stand-alone simple types.

<composite> element

<xs:complexType name="compositeDataType" mixed="true">

<xs:annotation><xs:documentation>

A derived data type; composed of two or more types

</xs:documentation></xs:annotation>

<xs:sequence>

<xs:element name="type" type="sbe:encodedDataType"

minOccurs="0" maxOccurs="unbounded" />

<xs:element name="composite"

type="sbe:compositeDataType"

minOccurs="0" maxOccurs="unbounded" />

</xs:sequence>

<xs:attribute name="name" type="sbe:symbolicName\_t"

use="required" />

<xs:attribute name="description" type="xs:string"

use="optional" />

<xs:attributeGroup ref="sbe:fieldAttributes" />

</xs:complexType>

Composite type example

In this example, a Price is encoded as 32 bit integer mantissa and a constant exponent, which is not sent on the wire.

<composite name="decimal32" semanticType="Price">

<type name="mantissa" primitiveType="int32" />

<type name="exponent" primitiveType="int8"

presence="constant">-4</type>

</composite>

### Enumeration encodings

An enumeration explicitly lists the valid values of a data domain. Any number of fields may share the same enumeration.

#### <enum> element

Each enumeration is represented by an <enum> element. It contains any number of <validValue> elements.

The encodingType attribute refers to a simple encoding of scalar type. The encoding of an enumeration may be char or any unsigned integer type.

| <enum> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of encoding | symbolic­­Name\_t | required | Must be unique among all encoding types. |
| description | Documentation of the type | string | optional |  |
| encodingType | Name of a simple encoding type | symbolic­­Name\_t | required | Must match the name attribute of a scalar <type> element. |

#### <validValue> element attributes

The name attribute of the <validValue> uniquely identifies it.

| <enum> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Symbolic name of value | symbolic­­Name\_t | required | Must be unique among valid values in the enumeration. |
| description | Documentation of the value | string | optional |  |
| sinceVersion | Documents the version of a schema in which a value was added | non­negative­Integer | default = 0 |  |

#### < validValue > element content

The element is required to carry a value, which is the valid value.

<xs:complexType name="enumType" mixed="true">

<xs:annotation><xs:documentation>

An enumeration of valid values

</xs:documentation></xs:annotation>

<xs:sequence>

<xs:element name="validValue" type="sbe:validValue"

minOccurs="1" maxOccurs="unbounded" />

</xs:sequence>

<xs:attribute name="name" type="sbe: symbolic­­Name\_t"

use="required" />

<xs:attribute name="encodingType" type="xs:string"

use="required" />

<xs:attribute name="description" type="xs:string" use="optional" />

</xs:complexType>

<xs:complexType name="validValue">

<xs:annotation><xs:documentation>

Valid value as a string

</xs:documentation></xs:annotation>

<xs:simpleContent>

<xs:extension base="xs:token">

<xs:attribute name="name" type="sbe:symbolic­­Name\_t" use="required" />

<xs:attribute name="description" type="xs:string"

use="optional" />

<xs:attribute name="sinceVersion" type="xs:nonNegativeInteger" default="0" />

</xs:extension>

</xs:simpleContent>

</xs:complexType>

Enumeration example (not all valid values listed)

This enumeration is encoded as an 8 bit unsigned integer value. Others are encoded as char codes.

<type name="intEnum" primitiveType="uint8" />

<enum name="PartyRole" encodingType="intEnum">

<validValue name="ExecutingFirm">1</validValue>

<validValue name="BrokerOfCredit”>2</validValue>

<validValue name="ClientID"3</validValue>

<validValue name="ClearingFirm”>4</validValue>

</enum>

### Multi-value choice encodings (bitset)

An enumeration explicitly lists the valid values of a data domain. Any number of fields may share the same set of choices.

#### <set> element

Each multi-value choice is represented by a <set> element. It may contain a number of <choice> elements up to the number of bits in the primitive encoding type. The largest number possible is 64 choices in a uint64 encoding.

The encodingType attribute refers to a simple encoding of scalar type. The encoding of a bitset should be an unsigned integer type.

| <set> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of encoding | symbolic­­Name\_t | required | Must be unique among all encoding types. |
| description | Documentation of the type | string | optional |  |
| encodingType | Name of a simple encoding type | string | required | Must match the name attribute of a scalar <type> element. |

#### <choice> element attributes

The name attribute of the <choice> uniquely identifies it.

| <choice> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Symbolic name of value | symbolic­­Name\_t | required | Must be unique among choices in the set. |
| description | Documentation of the value | string | optional |  |
| sinceVersion | Documents the version of a schema in which a choice was added | non­negative­Integer | default = 0 |  |

#### < choice > element content

The element is required to carry a value, which is the valid value.

<xs:complexType name="setType" mixed="true">

<xs:annotation>

<xs:documentation>

A multi value choice (bitset)

</xs:documentation>

</xs:annotation>

<xs:sequence>

<xs:element name="choice" type="sbe:choice"

minOccurs="1" maxOccurs="64" />

</xs:sequence>

<xs:attribute name="name" type="sbe: symbolic­­Name\_t"

use="required" />

<xs:attribute name="encodingType" type="xs:string" use="required" />

<xs:attribute name="description" type="xs:string" use="optional" />

</xs:complexType>

<xs:complexType name="choice">

<xs:annotation>

<xs:documentation>

A choice within a multi value set

Value is the position within a bitset (zero-based index).

</xs:documentation>

</xs:annotation>

<xs:simpleContent>

<xs:extension base="xs:nonNegativeInteger">

<xs:attribute name="name" type="sbe: symbolic­­Name\_t" use="required" />

<xs:attribute name="description" type="xs:string" use="optional" />

<xs:attribute name="sinceVersion" type="xs:nonNegativeInteger" default="0" />

</xs:extension>

</xs:simpleContent>

</xs:complexType>

Multi-value choice example The choice is encoded as a bitset.

<type name="bitset" primitiveType="uint8" />

<set name="Scope" encodingType="bitset" >

<choice name="LocalMarket">0</choice>

<choice name="National”>1</choice>

<choice name="Global"2</choice>

</set>

## Message template

To define a message type, add a <message> element to the root element of the XML document, <messageSchema>.

The name and id attributes are required. The first is a display name for a message, while the latter is a unique numeric identifier, commonly called template ID.

### Reserved space

By default, message size is the sum of its field lengths. However, a larger size may be reserved by setting blockLength, either to allow for future growth or for desired byte alignment. If so, the extra reserved space should be filled with zeros by message encoders.

### Message members

A <message> element contains its field definitions in three categories, which must appear in this sequence:

1. Element <field> defines a fixed-length field
2. Element <group> defines a repeating group
3. Element <data> defines a variable-length field, such as raw data

The number of members of each type is unbound.

### Member order

The order that fields are listed in the message schema governs the order that they are encoded on the wire.

**<message> element attributes**

| <message> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of a message | symbolic­­Name\_t | required |  |
| id | Unique message template identifier | unsignedInt | required | Must be unique within a schema |
| description | Documentation | string | optional |  |
| blockLength | Reserved size in number of octets for root level of message body | unsignedInt | optional | If specified, must be greater than or equal to the sum of field lengths. |
| semanticType | Documents value of FIX MsgType for a message | string | optional | Listed in FIX specifications |

Note that there need not be a one-to-one relationship between message template (identified by id attribute) and semanticType attribute. You might design multiple templates for the same FIX MsgType to optimize different scenarios.

<xs:element name="message">

<xs:annotation><xs:documentation>

A message type

</xs:documentation></xs:annotation>

<xs:complexType mixed="true">

<xs:sequence>

<xs:annotation><xs:documentation>

Fixed length fields only here

</xs:documentation></xs:annotation>

<xs:element name="field" type="sbe:fieldType" minOccurs="0" maxOccurs="unbounded" />

<xs:element name="group" type="sbe:groupType" minOccurs="0" maxOccurs="unbounded" />

<xs:element name="data" type="sbe:fieldType" minOccurs="0" maxOccurs="unbounded" />

</xs:sequence>

<xs:attribute name="name" type="sbe:symbolic­­Name\_t"

use="required" />

<xs:attribute name="id" type="xs:unsignedShort" use="required" />

<xs:attribute name="description" type="xs:string" use="optional" />

<!-- Space reserved for root level of message, not include groups -->

<xs:attribute name="blockLength" type="xs:nonNegativeInteger" use="optional" />

<!-- Standard MsgType value, such as 'D' for NewOrderSingle message -->

<xs:attribute name="semanticType" type="xs:string" use="optional" />

</xs:complexType>

</xs:element>

*Example <message> element*

<sbe:message name="NewOrderSingle" id="2" semanticType=”D”>

## Field attributes

Fields are added to a <message> element as child elements. See Field Encoding section above for a listing of all field types.

These are the common attributes of all field types.

| Schema attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of a field | symbolic­­Name\_t | required | Name and id must uniquely identify a field type within a message schema. |
| id | Unique field identifier (FIX tag) | unsignedShort | required |
| description | Documentation | string | optional |  |
| type | Encoding type name, one of simple type, composite type or enumeration. | string | required | Must match the name attribute of a simple <type>, <composite> encoding type or <enum>. |
| offset | Offset to the start of the field within a message or repeating group entry. By default, the offset is the sum of preceding field sizes, but it may be increased to effect byte alignment. | unsignedInt | optional | Must be greater than or equal to the sum of preceding field sizes. |
| presence | Field presence | enumera­tion | Default = required | required = field value is required; not tested for null.  optional = field value may be null; null value is default for data type.  constant= constant value not sent on wire. |
| sinceVersion | The version of a message schema in which this field was added. | non­negative­Integer | default=0 | Must not be greater than version attribute of <messageSchema> element. |

<xs:complexType name="fieldType">

<xs:annotation><xs:documentation>

A field of a message of a specified dataType

</xs:documentation></xs:annotation>

<xs:attribute name="name" type="sbe: symbolic­­Name\_t" use="required" />

<xs:attribute name="id" type="xs:unsignedShort" use="required" />

<!-- A simple or composite encoding type -->

<xs:attribute name="type" type="sbe: symbolic­­Name\_t" use="required" />

<xs:attribute name="sinceVersion" type="xs:nonnegativeInteger" default=”0” />

<xs:attribute name="description" type="xs:string" use="optional" />

<xs:attribute name="offset" type="xs:unsignedInt" use="optional" />

<xs:attributeGroup ref="sbe:fieldAttributes" />

<xs:attributeGroup ref="sbe:subFieldEncodingNames" />

</xs:complexType>

Example field schemas

Field that uses a composite encoding

<composite name="intQty32" semanticType="Qty">

<type name="mantissa" primitiveType="int32" />

<type name="exponent" primitiveType="int8" presence="constant">0</type>

</composite>

<field type="intQty32" name="OrderQty" id="38" offset="16"

description="Shares: Total number of shares" />

## Repeating group schema

A <group> has the same attributes as a <message> element. It has the same child members, with one exception: a repeating group may contain fields and nested repeating groups, but in may not contain variable-length <data> elements.

| <group> attribute | Description | XML type | Usage | Valid values |
| --- | --- | --- | --- | --- |
| name | Name of a group | symbolic­­Name\_t | required | Name and id must uniquely identify a group type within a message schema. |
| id | Unique group identifier | unsigned­Short | required |
| description | Documentation | string | optional |  |
| dimensionType | Dimensions of the repeating group | symbolic­Name\_t | default = groupSize­Encoding | If specified, must be greater than or equal to the sum of field lengths. |

<xs:complexType name="groupType">

<xs:sequence>

<xs:element name="field" type="sbe:fieldType"

minOccurs="0" maxOccurs="unbounded" />

<xs:element name="group" type="sbe:groupType" minOccurs="0" maxOccurs="unbounded" />

</xs:sequence>

<xs:attribute name="name" type="sbe:symbolicName\_t" use="optional" />

<xs:attribute name="id" type="xs:unsignedShort"

use="required" />

<xs:attribute name="description" type="xs:string"

use="optional" />

<xs:attribute name="blockLength"

type="xs:nonNegativeInteger" use="optional" />

<xs:attribute name="sinceVersion"

type="xs:nonNegativeInteger" default="0" />

<xs:attribute name="dimensionType" type="sbe:symbolicName\_t"

default="groupSizeEncoding" />

</xs:complexType>

*Example group schema with default dimension encoding*

<composite name="groupSizeEncoding">

<type name="blockLength" primitiveType="uint16"/>

<type name="numInGroup" primitiveType="uint16"

semanticType="NumInGroup"/>

</composite>

<group name="Parties" id="1012" >

<field type="string14" name="PartyID" id="448" />

<field type="partyRoleEnum" name="PartyRole" id="452" />

</group>

## Schema validation

The first level of schema validation is enforced by XML schema validation tools to make sure that a schema is well-formed according to XSD schema rules. Well-formed XML is necessary but insufficient to prove that a schema is correct according to FIX Simple Binary Encoding rules.

Additional conditions that render a schema invalid include the following.

| Error condition | Error description |
| --- | --- |
| Missing field encoding | A field or <enum> references a type name that is undefined. |
| Missing message header encoding | Missing encoding type for headerType specified in <messageSchema>. Default name is “messageHeader”. |
| Duplicate encoding name | An encoding name is non-unique, rendering a reference ambiguous. |
| nullValue specified for non-null encoding | Attribute nullValue is inconsistent with presence=required or constant |
| Attributes nullValue, minValue or maxValue of wrong data range | The specified values must be convertible to a scalar value consistent with the encoding. For example, if the primitive type is uint8, then the value must be in the range 0 through 255. |
| semanticType mismatch | If the attribute is specified on both a field and the encoding that it references, the values must be identical. |
| Missing constant value | If presence=constant is specified for a field or encoding, the element value must contain the constant value. |
| Missing validValue content | A <validValue> element is required to carry its value. |
| Incompatible offset and blockLength | A field offset greater than message or group blockLength is invalid |
| Duplicate ID or name of field or group | Attributes id and name must uniquely identify a type within a message schema. This applies to fields and groups. To be clear, the same field or group ID may be used in multiple messages, but each instance must represent the same type. Each of those instances must match on both id and name attributes. |

## Example message schemas

### Message with a repeating group

<message name="ListOrder" id="2" description="Simplified NewOrderList. Demonstrates repeating group">

<field name="ListID" id="66" type="string14" semanticType="String"/>

<field name="BidType" id="394" type="uint8" semanticType="int"/>

<group name="ListOrdGrp" id="2030" >

<field name="ClOrdID" id="11" type="string14" semanticType="String"/>

<field name="ListSeqNo" id="67" type="uint32" semanticType="int"/>

<field name="Symbol" id="55" type="string8" semanticType="String"/>

<field name="Side" id="54" type="char" ` semanticType="char"/>

<field name="OrderQty" id="38" type="intQty32" semanticType="Qty"/>

</group>

</message>

### Message with raw data fields

<message name="UserRequest" id="4" description="Demonstrates raw data usage">

<field name="UserRequestId" id="923" type="string14" semanticType="String"/>

<field name="UserRequestType" id="924" type="uint8" semanticType="int"/>

<field name="UserName" id="553" type="string14" semanticType="String"/>

<field name="Password" id="554" type="string14" semanticType="String"/>

<field name="NewPassword" id="925" type="string14" semanticType="String"/>

<field name="EncryptedPasswordMethod" id="1400" type="uint8" description="This should be an enum but values undefined." semanticType="int"/>

<field name="EncryptedPasswordLen" id="1401" type="uint8" semanticType="Length"/>

<field name="EncryptedNewPasswordLen" id="1403" type="uint8" semanticType="Length"/>

<field name="RawDataLength" id="95" type="uint8" semanticType="Length"/>

<data name="EncryptedPassword" id="1402" type="rawData" semanticType="data"/>

<data name="EncryptedNewPassword" id="1404" type="rawData" semanticType="data"/>

<data name="RawData" id="96" type="rawData" semanticType="data"/>

</message>

## Reserved element names

### Composite types

| Encoding type name (default names) |
| --- |
| messageHeader |
| groupSizeEncoding |

### Composite type elements

| Type name | Composite type |
| --- | --- |
| blockLength | messageHeader and groupSize |
| day | MonthYear |
| exponent | decimal |
| mantissa | decimal |
| month | MonthYear |
| numInGroup | groupSize |
| templateId | messageHeader |
| time | TZ time |
| timezoneHour | TZ time |
| timezoneMinute | TZ time |
| version | messageHeader |
| week | MonthYear |
| year | MonthYear |

# Schema Extension Mechanism

## Objective

It is not always practical to update all message publishers and consumers simultaneously. Within certain constraints, message schemas and wire formats can be extended in a controlled way. Consumers using an older version of a schema should be compatible if interpretation of added fields or messages is not required for business processing.

### Constraints

Compatibility is only ensured under these conditions:

* Fields may be added to either the root of a message or to a repeating group, but in each case, they must be appended to end of a block.
* Existing fields cannot change data type or move within a message.
* A repeating group may be added, but only after existing groups.

Changes that break those constraints require consumers to update to the current schema used by publishers.

## Message schema features for extension

### Schema version

The <messageSchema> root element contains a version number attribute. By default, version is zero, the initial version of a message schema. Each time a message schema is changed, the version number is incremented.

Version applies to the schema as a whole, not to individual elements. Version is sent in the message header so the consumer can determine which version of the message schema was used to encode the message.

See section 4.2.1 above for schema attributes.

### Since version

When a new field, group or message is added to a message schema, the extension may be documented by adding a sinceVersion attribute to the element. The sinceVersion attribute tells in which schema version the element was added. This attribute remains the same for that element for the lifetime of the schema. This attribute is for documentation purposes only, it is not sent on the wire.

Over time, multiple extensions may be added to a message schema. New fields must be appended following earlier extensions. By documenting when each element was added, it possible to verify that extensions were appended in proper order.

### Block length

The length of root level of the message may optionally be documented on a <message> element in the schema using the blockLength attribute. See section 4.4.1 above for message attributes. If not set in the schema, block length of the message root is the sum of its field lengths. Whether it is set in the schema or not, the block length is sent on the wire to consumers.

Likewise, a repeating group has a blockLength attribute to tell how much space is reserved for group entries, and the value is sent on the wire. It is encoded in the schema as part of the NumInGroup field encoding. See section 2.13.2 above.

## Wire format features for extension

### Message size

It is assumed that the session protocol header conveys the size of the whole message. See section 3.1 above. This enables a consumer to properly frame messages even when the message has been lengthened in a later version of the schema.

### Block size

The length of the root level of the message is sent on the wire in the SBE message header. See section 3.2.3 above. Therefore, if new fields were appended in a later version of the schema, the consumer would still know how many octets to consume to find the next message element, such as repeating group or variable-length Data field. Without the current schema version, the consumer cannot interpret the new fields, but it does not break parsing of earlier fields.

Likewise, block size of a repeating group is conveyed in the NumInGroup encoding.

## Message schema extension example

Initial version of a message schema

<messageSchema package="FIXBinaryTest"

byteOrder="littleEndian">

<types>

<type name="int8" primitiveType="int8"/>

</types>

<message name="FIX Binary Message1" id="1" blockLength="4">

<field name="Field1" id="1" type="int8" semanticType="int"/>

</message>

</messageSchema>

Second version – a new message is added

<messageSchema package="FIXBinaryTest" byteOrder="littleEndian"

version="1">

<types>

<type name="int8" primitiveType="int8"/>

<type name="int16" primitiveType="int16"

sinceVersion="1"/>

</types>

<message name="FIX Binary Message1" id="1" blockLength="4">

<field name="Field1" id="1" type="int8" semanticType="int"/>

</message>

<!-- New message added in this version-->

<message name="FIX Binary Message2" id="2" blockLength="4" sinceVersion="1">

<field name="Field2" id="2" type="int16" semanticType="int"/>

</message>

</messageSchema>

Third version – a field is added

<messageSchema package="FIXBinaryTest" byteOrder="littleEndian"

version="2">

<types>

<type name="int8" primitiveType="int8"/>

<type name="int16" primitiveType="int16"

sinceVersion="1"/>

<type name="int32" primitiveType="int32"

sinceVersion="2"/>

</types>

<message name="FIX Binary Message1" id="1" blockLength="8">

<field name="Field1" id="1" type="int8" semanticType="int"/>

<field name="Field11" id="11" type="int32" semanticType="int"

sinceVersion="2"/>

</message>

<message name="FIX Binary Message2" id="2" blockLength="4"

sinceVersion="1">

<field name="Field2" id="2" type="int16" semanticType="int"/>

</message>

</messageSchema>

# Usage Guidelines

## Identifier encodings

FIX specifies request and entity identifiers as String type. Common practice is to specify an identifier field as fixed-length character of a certain size.

Optionally, a message schema may restrict such identifiers to numeric encodings.

Example of an identifier field with character encoding

<type name="idString" primitiveType="char" length="16" />

<field name="QuoteReqId" id="131" type="idString" semanticType="String"/>

Example of an identifier field with numeric encoding

<type name="uint64" primitiveType="uint64" />

<field name="QuoteReqId" id="131" type="uint64" semanticType="String"/>

# Examples

Not all FIX enumeration values are listed in the samples.

## Flat, fixed-length message

This is an example of a simple, flat order message without repeating groups or variable-length data.

### Sample order message schema

<sbe:messageSchema xmlns:sbe="http://www.fixprotocol.org/ns/simple/1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" package="Examples" description="Test dictionary" byteOrder="littleEndian" xs:schemaLocation="http://www.fixprotocol.org/ns/simple/1.0/SimpleBinary1-0.xsd">

<types>

<type name="enumEncoding" primitiveType="char"/>

<type name="idString" length="8" primitiveType="char" semanticType="String"/>

<type name="timestampEncoding" primitiveType="uint64"

semanticType="UTCTimestamp"/>

<composite name="messageHeader"

description="Template ID and length of message root">

<type name="blockLength" primitiveType="uint16" semanticType="Length"/>

<type name="templateId" primitiveType="uint16"/>

<type name="version" primitiveType="uint8"/>

<type name="reserved" primitiveType="uint8"/>

</composite>

<composite name="optionalDecimalEncoding"

description="Optional decimal with constant exponent">

<type name="mantissa" presence="optional" primitiveType="int64"/>

<type name="exponent" presence="constant" primitiveType="int8">-3</type>

</composite>

<composite name="qtyEncoding" description="Decimal constrainted to integers">

<type name="mantissa" primitiveType="int32"/>

<type name="exponent" presence="constant" primitiveType="int8">0</type>

</composite>

<enum name="ordTypeEnum" encodingType="enumEncoding">

<validValue name="Market" description="Market">1</validValue>

<validValue name="Limit" description="Limit">2</validValue>

<validValue name="Stop" description="Stop Loss">3</validValue>

<validValue name="StopLimit" description="Stop Limit">4</validValue>

</enum>

<enum name="sideEnum" encodingType="enumEncoding">

<validValue name="Buy" description="Buy">1</validValue>

<validValue name="Sell" description="Sell">2</validValue>

</enum>

</types>

<sbe:message name="NewOrderSingle" id="99" blockLength="54" semanticType="D">

<field name="ClOrdId" id="11" type="idString" description="Customer Order ID"

offset="0" semanticType="String"/>

<field name="Account" id="1" type="idString" description="Account mnemonic"

offset="8" semanticType="String"/>

<field name="Symbol" id="55" type="idString" description="Security ID"

offset="16" semanticType="String"/>

<field name="Side" id="54" type="sideEnum" description="Side" offset="24"

semanticType="char"/>

<field name="TransactTime" id="60" type="timestampEncoding"

description="Order entry time" offset="25" semanticType="UTCTimestamp"/>

<field name="OrderQty" id="38" type="qtyEncoding" description="Order quantity"

offset="33" semanticType="Qty"/>

<field name="OrdType" id="40" type="ordTypeEnum" description="Order type"

offset="37" semanticType="char"/>

<field name="Price" id="44" type="optionalDecimalEncoding"

description="Limit price" offset="38" semanticType="Price"/>

<field name="StopPx" id="99" type="optionalDecimalEncoding"

description="Stop price" offset="46" semanticType="Price"/>

</sbe:message>

</sbe:messageSchema>

**Notes on the message schema**

In this case, there is a lot of verbiage for one message, but in practice, a schema would define a set of messages. The same encodings within the <types> element would be used for a whole collection of messages. For example, a price encoding need only be defined once but can be used in any number of messages in a schema. Many of the attributes, such as description, offset, and semanticType, are optional but are shown here for a full illustration.

All character fields in the message are fixed-length. Values may be shorter than the specified field length, but not longer. Since all fields are fixed-length, they are always in a fixed position, supporting direct access to data.

An enumeration gives the valid values of a field. Both enumerations in the example use character encoding, but note that some enumerations in FIX are of integer type.

There are two decimal encodings. The one used for quantity sets the exponent to constant zero. In effect there is no fractional part and only the mantissa is sent on the wire, acting as an integer. However, FIX defines Qty as a float type since certain asset classes may use fractional shares.

The other decimal encoding is used for prices. The exponent is constant -3. In essence, each price is transmitted as an integer on the wire with assumed three decimal places. Each of the prices in the message is conditionally required. If OrdType=Limit, then Price field required. If OrdType=Stop then StopPx is required. Otherwise, if OrdType=Market, then neither price is required. Therefore, the price takes an optional encoding. To indicate that it is null, a special value is sent on the wire. See the table in section 2.4.2 above for the null value of the int64 mantissa.

In this example, all fields are packed without special byte alignment. Performance testing may prove better results with a different arrangement of the fields or adjustments to field offsets. However, those sorts of optimizations are platform dependent.

### Wire format of an order message

Hexadecimal and ASCII representations (little-endian byte order):

3e 00 36 00 63 00 00 00 4f 52 44 30 30 30 30 31 :> 6 c ORD00001

41 43 43 54 30 31 00 00 47 45 4d 34 00 00 00 00 :ACCT01 GEM4

31 c0 21 ed 1b 04 c3 2b 13 07 00 00 00 32 1a 85 :1 ! + 2

01 00 00 00 00 00 ff ff ff ff ff ff ff 7f :

**Interpretation**

| **Field ID** | **Name** | **Offset**  root block | **Length** | **Interpreted value** |
| --- | --- | --- | --- | --- |
| 11 | ClOrdId | 0 | 8 | ORD00001 |
| 1 | Account | 8 | 8 | ACCT01 |
| 55 | Symbol | 16 | 8 | GEM4 |
| 54 | Side | 24 | 1 | 1 Buy |
| 60 | TransactTime | 25 | 8 | 2013-10-10 13:35:33.135 |
| 38 | OrderQty | 33 | 4 | 7 |
| 40 | OrdType | 37 | 1 | 2 Limit |
| 44 | Price | 38 | 8 | 99.610 |
| 99 | StopPx | 46 | 8 | null |

The first uint16 contains message size, decimal 62. Message size includes the header and body of the message.

The next 6 octets contain the SBE message header: Block length=54 Template ID=99 Schema version=0. Block length includes the size of the root of the message. That is, the message body. Template ID and schema version are from the message schema. Template ID is from the “id” attribute of the message, and schema version zero is a default value.

Next follows three character fields for ClOrdId, Account and Symbol. Note that each is padded with ASCII NUL (zero) to the field length of 8 characters. This is controlled by encoding idString in the message schema.

At offset 24 from the start of the message body (offset 32 from the start of the buffer) is “1” value of the Side field, meaning “Buy”. See the first entry of sideEnum for that valid value of the Side.

Next at offset 25 is a uint64 value representing a timestamp as nanoseconds since the UNIX epoch.

OrdQty value 7 is at offset 33.

OrdType value “2” at offset 37 means “Limit”. See ordTypeEnum in the XML schema.

Price at offset 38 is the int64 value 1851A, or decimal 99610. Since the constant exponent is -3, this represents the price 99.610.

The StopPx field at offset 46 contains the null value of int64 since this is not a stop order.

## Message with repeating group

This is an example of a message with a repeating group.

### Sample execution report message schema

Add this encoding types to those in the previous example.

<types>

<type name="date" primitiveType="uint16" semanticType="LocalMktDate"/>

<composite name="MONTH\_YEAR" semanticType="MonthYear">

<type name="year" primitiveType="uint16"/>

<type name="month" primitiveType="uint8"/>

<type name="day" primitiveType="uint8"/>

<type name="week" primitiveType="uint8"/>

</composite>

<composite name="groupSizeEncoding" description="Repeating group dimensions">

<type name="blockLength" primitiveType="uint16" semanticType="Length"/>

<type name="numInGroup" primitiveType="uint16" semanticType="NumInGroup"/>

</composite>

<enum name="execTypeEnum" encodingType="enumEncoding">

<validValue name="New" description="New">0</validValue>

<validValue name="DoneForDay" description="Done for day">3</validValue>

<validValue name="Canceled" description="Canceled">4</validValue>

<validValue name="Replaced" description="Replaced">5</validValue>

<validValue name="PendingCancel" ">6</validValue>

<validValue name="Rejected" description="Rejected">8</validValue>

<validValue name="PendingNew" description="Pending New">A</validValue>

<validValue name="Trade" description="partial fill or fill">F</validValue>

</enum>

<enum name="ordStatusEnum" encodingType="enumEncoding">

<validValue name="New" description="New">0</validValue>

<validValue name="PartialFilled>1</validValue>

<validValue name="Filled" description="Filled">2</validValue>

<validValue name="DoneForDay" description="Done for day">3</validValue>

<validValue name="Canceled" description="Canceled">4</validValue>

<validValue name="PendingCancel>6</validValue>

<validValue name="Rejected" description="Rejected">8</validValue>

<validValue name="PendingNew" description="Pending New">A</validValue>

<validValue name="PendingReplace" >E</validValue>

</enum>

</types>

<sbe:message name="ExeuctionReport" id="98" blockLength="42" semanticType="8">

<field name="OrderID" id="37" type="idString" description="Order ID"

offset="0" semanticType="String"/>

<field name="ExecID" id="17" type="idString" description="Execution ID"

offset="8" semanticType="String"/>

<field name="ExecType" id="150" type="execTypeEnum"

description="Execution type" offset="16" semanticType="char"/>

<field name="OrdStatus" id="39" type="ordStatusEnum"

description="Order status" offset="17" semanticType="char"/>

<field name="Symbol" id="55" type="idString" description="Security ID"

offset="18" semanticType="String"/>

<field name="MaturityMonthYear" id="200" type="MONTH\_YEAR"

description="Expiration" offset="26" semanticType="MonthYear"/>

<field name="Side" id="54" type="sideEnum" description="Side" offset="31"

semanticType="char"/>

<field name="LeavesQty" id="151" type="qtyEncoding"

description="Quantity open" offset="32" semanticType="Qty"/>

<field name="CumQty" id="14" type="qtyEncoding"

description="Executed quantity" offset="36" semanticType="Qty"/>

<field name="TradeDate" id="75" type="date"

description="Trade date" offset="40" semanticType="LocalMktDate"/>

<group name="FillsGrp" id="2112" description="Partial fills" blockLength="12"

dimensionType="groupSizeEncoding">

<field name="FillPx" id="1364" type="optionalDecimalEncoding"

description="Price of partial fill" offset="0" semanticType="Price"/>

<field name="FillQty" id="1365" type="qtyEncoding"

description="Executed quantity" offset="8" semanticType="Qty"/>

</group>

</sbe:message>

**Notes on the message schema**

The message contains a MonthYear field. It is encoded as a composite type with year, month, day and week subfields.

This message layout contains a repeating group containing a collection of partial fills for an execution report. The <group> XML tag enclosed the fields within a group entry. The dimensions of the repeating group are encoding as a composite type called groupSizeEncoding.

### Wire format of an exeucution message

Hexadecimal and ASCII representations (little-endian byte order):

4e 00 2a 00 62 00 00 00 4f 30 30 30 30 30 30 31 :N \* b O0000001

45 58 45 43 30 30 30 30 46 31 47 45 4d 34 00 00 :EXEC0000F1GEM4

00 00 de 07 06 ff ff 31 01 00 00 00 06 00 00 00 : 1

75 3e 0c 00 02 00 1a 85 01 00 00 00 00 00 02 00 :u>

00 00 24 85 01 00 00 00 00 00 04 00 00 00 00 00 : $

**Interpretation**

| **Field ID** | **Name** | **Offset**  in block | **Length** | **Interpreted value** |
| --- | --- | --- | --- | --- |
| 37 | OrderID | 0 | 8 | O0000001 |
| 17 | ExecID | 8 | 8 | EXEC0000 |
| 150 | ExecType | 16 | 1 | F Trade |
| 39 | OrdStatus | 17 | 1 | 1 PartialFilled |
| 55 | Symbol | 18 | 8 | GEM4 |
| 200 | MaturityMonthYear | 26 | 5 | 201406 |
| 54 | Side | 31 | 1 | 1 Buy |
| 151 | LeavesQty | 32 | 4 | 1 |
| 14 | CumQty | 36 | 4 | 6 |
| 75 | TradeDate | 40 | 2 | 2013-10-11 |
| 2112 | FillsGrp |  |  | instance 0 |
| 1364 | FillPx | 0 | 8 | 99.610 |
| 1365 | FillQty | 8 | 4 | 2 |
| 2112 | FillsGrp |  |  | instance 1 |
| 1364 | FillPx | 0 | 8 | 99.620 |
| 1365 | FillQty | 8 | 4 | 4 |

The first uint16 contains message size, decimal 78.

The next 6 octets contain the SBE message header: Block length=42 Template ID=98 Schema version=0. Block length includes the size of the root of the message, not including the repeating group.

The groupSizeEncoding is at offset 50 in the buffer. It contains a uint16 containing the length of each repeating group entry and the count of entries as another uint16. The block length of the repeating group is 12 and its NumInGroup value is 2.

The first entry of the repeating group immediately follows the groupSizeEncoding on the wire. Each of the two entries occupies 12 octets.

In the MonthYear encoding, only the year and month subfields are populated. The day and week subfields are populated with the null value for uint8 type, 255 (hex ff).