FIX TRADING COMMUNITY™
INDUSTRY-DRIVEN • INDEPENDENT • NEUTRAL

FIX Simple Binary Encoding

Technical Specification

Working Draft for potential Release Candidate 4

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.

Contents

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 RFC2119. 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 but 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

Related FIX Standards

Simple Open Framing Header, FIX Protocol, Limited. Release Candidate 1 specification has been published at http://www.fixtradingcommunity.org/

For FIX semantics, see the current FIX message specification, which is currently FIX 5.0 Service Pack 2 with Extension Packs.

Dependencies on other standards

SBE is dependent on several industry standards. Implementations must conform to these standards to interoperate. Therefore, they are normative for SBE.

IEEE 754-2008 A
Standard for Binary Floating-Point Arithmetic

ISO 639-1:2002
Codes for the representation of names of languages - Part 1: Alpha-2 code

ISO 3166-1:2013
Codes for the representation of names of countries and their subdivisions - Part 1: Country codes

ISO 4217:2008
Codes for the representation of currencies and funds

ISO 8601:2004
Data elements and interchange formats - Information interchange - Representation of dates and times

ISO 10383:2012
Securities and related financial instruments - Codes for exchanges and market identification (MIC)

XML 1.1 schema standards are located here W3C XML Schema

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 on 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 primitive 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 instrument.
- 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 which case, the data is not sent on the wire, but may be treated as constants by applications.

Default value

Default value handling is not specified by the encoding layer. A null value of an optional field does not necessarily imply that a default value should be applied. Rather, default handling is left to application layer specifications.

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 field 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	2.5	A real number with binary representation of specified precision
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	2.7.1	Single US-ASCII character value. Can include any alphanumeric character or punctuation. All char fields are case sensitive (i.e. m != M).
String	Fixed-length character array	2.7.2	A fixed-length character array of ASCII encoding
String	Variable-length data encoding	2.7.3	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	2.7.3	Non-ASCII string. The character encoding may be specified by a schema attribute.
XMLData	String encoding	2.7.3	Variable-length XML. Must be paired with a Length field.
data	Fixed-length data	2.8.1	Fixed-length non-character data
data	Variable-length data encoding	2.8.2	Variable-length data. Must be paired with a Length field.
Country	Fixed-length character array; size =	2.7.2	ISO 3166-1:2013 Country code

	2 or a subset of values may use Enumeration encoding		
Currency	Fixed-length character array; size = 3 or a subset of values may use Enumeration encoding	2.7.2	ISO 4217:2008 Currency code (3 character)
Exchange	Fixed-length character array; size = 4 or a subset of values may use Enumeration encoding	2.7.2	ISO 10383:2012 Market Identifier Code (MIC)
Language	Fixed-length character array; size = 2 or a subset of values may use Enumeration encoding	2.7.2	National language - uses ISO 639-1:2002 standard
Implicit enumeration—char or int	Enumeration encoding	2.12	A single choice of alternative values
Boolean	Boolean encoding	2.12.6	Values true or false
MultipleCharValue	Multi-value choice encoding	2.13	Multiple choice of a set of values
MultipleStringValue	Multi-value choice encoding. String choices must be mapped to int values.	2.13	Multiple choice of a set of values
MonthYear	MonthYear encoding	2.8	A flexible date format that must include month and year at least, but may also include day or week.
UTCTimestamp	Date and time encoding	2.9	Time/date combination represented in UTC (Universal Time Coordinated, also known as "GMT")
UTCTimeOnly	Date and time encoding	2.9	Time-only represented in UTC (Universal Time Coordinated, also known as "GMT")
UTCDateOnly	Date and time encoding	2.9	Date represented in UTC (Universal Time Coordinated, also known as "GMT")
LocalMktDate	Local date encoding	2.9	Local date(as oppose to UTC)

TZTimeOnly	TZTimeOnly	2.11.3	Time of day
TZTimestamp	TZTimestamp	2.11.1	Time/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 8601:2004

The FIX semantic types listed above are spelled and capitalized exactly as they are in the FIX repository from which official FIX documents and references are derived.

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 need not be populated. A special null value indicates that a field is not set. The presence attribute may be specified on either on a field or its encoding.
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.

Non-FIX types

Encodings may be added to SBE messages that do not correspond to listed FIX data types. In that case, the encoding and fields that use the encoding will not have a semanticType attribute.

Integer encoding

Integer encodings should be used for cardinal or ordinal number fields. Signed integers are encoded in a two's complement binary format.

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.

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 4.3.1 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 mantiss, 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.

Schema attribute	decimal	decimal64	decimal32
exponent range	–128 to 127	–128 to 127	–128 to 127
mantissa range	–263 + 1 to 263 – 1	–263 + 1 to 263 – 1	–231 + 1 to 231 – 1
minValue	(–263 + 1) * 10127	(–263 + 1) * 10127	(–231 + 1) * 10127
maxValue	(263 – 1) * 10127	(–263 – 1) * 10127	(231 – 1) * 10127
nullValue	mantissa=–263, exponent=–128	mantissa =–263	mantissa =–231

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 padding

When both mantissa and exponent are sent on the wire for a decimal, the elements are packed by default. However, byte alignment may be controlled by specifying offset of the exponent within the composite encoding. See section 4.4.4.3 below.

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 754-2008). 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 754-2008 standard for ranges and details of the encodings.

Primitive type	Description	IEEE 754-2008 format	Length (octets)
float	Single precision floating point	binary32	4
double	Double precision floating point	binary64	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 4.3.1 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 character strings that cannot be constrained to a specific size. It may also be used for non-ASCII encoded strings.

Character

Character fields hold a single character. They are most commonly used for field with character code enumerations. See section 2.12 below for discussion of enum fields.

FIX data type	Description	Backing primitive	Length (octet)
char	A single US-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).

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)

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	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.

Range attributes minValue and maxValue do not apply to fixed-length character arrays.

US-ASCII is the default encoding of character arrays to conform to usual FIX values. The characterEncoding attribute may be specified to override encoding.

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 string encoding

Variable-length string encoding is used for variable length ASCII strings or embedded non-ASCII character data (like EncodedText field). A separate length field coveys the size of the field.

On the wire, length immediately precedes the data.

The length subfield may not be null, but may be set to zero for an empty string. In that case, no space is reserved for the data. No distinction is made at an encoding layer between an empty string and a null string. Semantics of an empty variable-length string should be specified at an application layer.

FIX data type	Description	Backing primitives	Length (octets)
Length	The length of variable data in octets	primitiveType="uint8" or "uint16" May not hold null value.	1 or 2
data	Raw data	Array of octet of size specified in associated Length field. The data field itself should be specified as variable length. primitiveType="uint8" length="0" indicates variable length	variable

Optionally, implementations may support any other unsigned integer types for length.

Range attributes for string Length

Schema attribute	length uint8	length uint16	data
minValue	0	0	N/A
maxValue	254	65534	N/A

If the Length element has minValue and maxValue attributes, it specifies the minimum and maximum *length* of the variable-length data.

Range attributes minValue , maxValue, and nullValue do not apply to the data element.

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

Encoding specifications for variable-length string

Variable length string is encoded as a 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="varString" description="Variable-length string">
    <type name="length" primitiveType="uint16" semanticType="Length"/>
    <type name="data" length="0" primitiveType="uint8"
    semanticType="data" characterEncoding="UTF-16"/>
</composite>

<data name="SecurityDesc" id="107" type="varString"/>
```

The characterEncoding attribute tells which variable-sized encoding is used if the data field represents encoded text. UTF-8 is the recommended encoding, but there is no default in the XML schema

Example of a variable-length string field

Example shows encoded bytes on the wire.

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

M S F T

```
04004d534654
```

Data encodings

Raw data is opaque to SBE. In other words, it is not constrained by any value range or structure known to the messaging layer other than length. Data fields simply convey arrays of octets.

Data may either be of fixed-length or variable-length. In Simple Binary Encoding, fixed-length data encoding may be used for data of predetermined length, even though it does not represent a FIX data type. Variable-length encoding should be reserved for raw data when its length is not known until run-time.

Fixed-length data

Data arrays are allocated as a fixed space in a message, supporting direct access to fields. A fixed size array is distinguished from a variable length data by the presence of a length schema attribute rather than sending length on the wire.

FIX data type	Description	Backing primitives	Length (octets)	Required schema attribute
data	octet array	Array of uint8 of specified length.	Specified by length attribute	length

Encoding specifications for fixed-length data

A fixed-length octet array encoding should specify primitiveType="uint8" and a length attribute is required.

Data range attributes minValue and maxValue do not apply.

Since raw data is not constrained to a character set, characterEncoding attribute should not be specified.

Example of fixed-length data encoding

A fixed-length data encoding specification for a binary user ID

```
<type name="uuid" primitiveType="uint8" length="16" description="RFC 4122 compliant UUID"/>

<field type="uuid" name="Username" id="553" />
```

Variable-length data encoding

Variable-length data is used for variable length non-character data (such as RawData). A separate length field conveys the size of the field. On the wire, length immediately precedes the data.

The length subfield may not be null, but it may be set to zero. In that case, no space is reserved for the data. Semantics of an empty variable-length data element should be specified at an application layer.

FIX data type	Description	Backing primitives	Length (octets)
Length	The length of variable data in octets	primitiveType="uint8" or "uint16" May not hold null value.	1 or 2
data	Raw data	Array of octet of size specified in associated Length field. The data field itself should be	variable

specified as variable length.
primitiveType="uint8"

Optionally, implementations may support any other unsigned integer types for length.

Range attributes for variable-length data Length

Schema attribute	length uint8	length uint16	data
minValue	0	0	N/A
maxValue	254	65534	N/A

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 , maxValue, and nullValue 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.

Encoding specifications for variable-length data

Variable length data is encoded as composite type, consisting of a length sub field and data subfield.

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="data" 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

```
04004d534654
```

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 padding

The four subfields of MonthYear are packed at an octet level by default. However, byte alignment may be controlled by specifying offset of the elements within the composite encoding. See section 4.4.4.3 below.

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

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. Time unit may be serialized on the wire if timestamps are of mixed precision. On the other hand, if all timestamps have the same precision, then time unit may be set to a constant in the message schema. Then it need not be sent on the wire.

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 time	8	epoch=”unix” (default)
	timeUnit = second or millisecond or microsecond or nanosecond May be constant	uint8 unit	1	
UTCTimeOnly	UTC time of day only Default: nanoseconds since midnight today	uint64 time	8	
	timeUnit = second or millisecond or microsecond or nanosecond May be constant	uint8 unit	1	
UTCDateOnly	UTC calendar date Default: days since Unix epoch. Range: Jan. 1, 1970 - June 7, 2149	uint16	2	epoch=”unix” (default)

Encoding specifications for date and time

Time specifications use an enumeration of time units. See section 2.13 below for a fuller explanation of enumerations.

Enumeration of time units:

```
<enum name="TimeUnit" encodingType="uint8">
    <validValue name="second">0</validValue>
    <validValue name="millisecond">3</validValue>
    <validValue name="microsecond">6</validValue>
   <validValue name="nanosecond">9</validValue>
</enum>
```

Timestamp with variable time units:

```
<composite name="UTCTimestamp" description="UTC timestamp with precision on
the wire" semanticType="UTCTimestamp" >
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" />
</composite>
```

Timestamp with constant time unit:

```
<composite name="UTCTimestampNanos" description="UTC timestamp with
nanosecond precision" semanticType="UTCTimestamp" >
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" presence="constant"
valueRef="TimeUnit.nanosecond" />
</composite>
```

Time only with variable time units:

```
<composite name="UTCTime" description="Time of day with precision on the
wire" semanticType="UTCTimeOnly" >
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" />
</composite>
```

Time only with constant time unit:

```
<composite name="UTCTimeNanos" description="Time of day with millisecond
precision" semanticType="UTCTimeOnly" >
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" presence="constant"
valueRef="TimeUnit.millisecond" />
</composite>
```

Date only specification:

```
<type name="date" primitiveType="uint16" semanticType="UTCDateOnly" />
```

Examples of date/time fields

timestamp 14:17:22 Friday, October 4, 2024 UTC (20,000 days and 14 hours, 17 minutes and 22 seconds since the UNIX epoch) with default schema attributes

```
<composite name="UTCTimestampNanos" description="UTC timestamp with
nanosecond precision" semanticType="UTCTimestamp" >
<type name="time" primitiveType="uint64" />
<type name="unit" primitiveType="uint8" presence="constant"
valueRef="TimeUnit.nanosecond" />
</composite>
```

Wire format of UTCTimestamp with constant time unit in little-Endian byte order

```
4047baa145fb17
```

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

```
<composite name="UTCTimeOnlyNanos" description="UTC time of day with
nanosecond precision" semanticType="UTCTimeOnly" >
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" presence="constant"
valueRef="TimeUnit.nanosecond" />
</composite>
```

Wire format of UTCTimeOnly

```
10d74916220000
```

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	uint16	2	epoch=”unix” (default)

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 8601:2004 format ±hhmm.

TZTimestamp encoding

A binary UTCTimestamp followed by a number representing the time zone indicator as defined in ISO 8601:2004.

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

Composite encoding padding

The subfields of TZTimestamp are packed at an octet level by default. However, byte alignment may be controlled by specifying offset of the elements within the composite encoding. See section 4.4.4.3 below.

Standard TZTimestamp encoding specification

```
<composite name="tzTimestamp" semanticType="TZTimestamp">
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" />
    <!-- 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 8601:2004.

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 Default: nanoseconds since midnight today, local time	uint64	8	None
	timeUnit = second or millisecond or microsecond or nanosecond May be constant	uint8	1	None
	Time zone hour offset	int8	1	None
	Time zone minute offset	uint8	1	None

Composite encoding padding

The subfields of TZTimeOnly are packed at an octet level by default. However, byte alignment may be controlled by specifying offset of the elements within the composite encoding. See section 4.4.4.3 below.

Standard TZTimeOnly encoding specification

```
<composite name="tzTimeOnly" semanticType="TZTimeOnly">
    <type name="time" primitiveType="uint64" />
    <type name="unit" primitiveType="uint8" />
    <!-- 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. Implementations must support char and uint8 types. They may additionally support other unsigned integer types to allow more 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 US-ASCII printable characters. See section 2.7.1 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.

Enumerations of other data types, such as String valid values specified in FIX, should be mapped to an integer wire format in SBE.

Encoding specification of enumeration

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.

An `<enum>` element must have an encodingType attribute to specify the type of its values. Two formats of encodingType are acceptable:

- In-line style: the value of encodingType is its primitive data type.
- Reference style: the value of encodingType is the name of a `<type>` element that specifies the wire format.

The length of a `<type>` associated to an enumeration must be 1. That is, enumerations should only be backed by scalar types, not arrays.

Enumeration examples

These examples use a char field for enumerated code values.

Example enum lists acceptable values and gives the underlying encoding, which in this case is char (in-line style)

```
<enum name="SideEnum" 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>
```

Reference to type: This specification is equivalent to the one above.

```
<type name="charEnumType" primitiveType="char"/>
    <enum name="SideEnum" encodingType="charEnumType">
    <!-- valid values as above -->
</enum>
```

Side field specification references the enumeration type

```
<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="PartyIDSourceEnum.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

```
<enum name="booleanEnum" 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 bitset (a fixed-size sequence of bits, also known as bitmap, bit array or bit vector) to represent up to 64 possible choices. The encoding is backed by an unsigned integer. 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

Like other integer-backed encodings, multi-value encodings follow the byte order specified by message schema when serializing to the wire. See section 4.3.1 for specification of message schema attributes, including byteOrder.

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.

Encoding specification of multi-value choice

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.

A `<set>` element must have an encodingType attribute to specify the wire format of its values. Two formats of encodingType are recognized :

- In-line style: the value of encodingType is its primitive data type.
- Reference style: the value of encodingType is the name of a `<type>` element that specifies the wire format.

The length of a `<type>` associated to an bitset must be 1. That is, bitsets should not be specified as arrays.

Multi-value example

Example of a multi-value choice (was MultipleCharValue in tag-value encoding) Encoding type is
in-line style.

```
<set name="FinancialStatusEnum" encodingType="uint8">
    <choice name="Bankrupt">0</choice>
    <choice name="Pending delisting">1</choice>
    <choice name="Restricted">2</choice>
</set>
```

Reference to type. This is equivalent to the example above.

```
<type name="u8Bitset" primitiveType="uint8"/>

<set name="FinancialStatusEnum" encodingType="u8Bitset">
<!--choices as above -->
</set>
```

A field using the multi-choice encoding

```
<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-US-ASCII printable characters or other invalid sequence if a different characterEncoding is specified.
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

Message Framing

SBE messages need framing when used with protocols that do not preserve message boundaries, such as when they are transmitted on a streaming session protocol or are persisted in storage. Be aware that framing features may or may not be encoded in SBE.

Simple Open Framing Header

FIX Protocol Ltd. offers the Simple Open Framing Header standard for framing messages encoded with binary wire formats, such as Simple Binary Encoding.

The framing header provides two features:

- An overall message size including headers to support framing
- An identifier of the encoding used in the message payload. This supports selecting the correct decoder in the case where multiple message encodings are used on a session. It also aids tooling such as protocol analyzers to identify message protocols contained in network packets.

While the Simple Open Framing Header specification is normative, the following is an interpretation of that standard as an SBE encoding. Note that the framing standard specifies that the framing header will always be encoded in big-endian byte order, also known as network byte order.

Simple Open Framing Header as an SBE composite encoding (big-endian)

```
<composite name="framingHeader"/>
    <type name="messageLength" primitiveType="uint32" />
    <type name="encodingType" primitiveType="uint16" />
</composite>
```

The values of encodingType used to indicate SBE payloads are currently defined as:

Encoding	encodingType value
SBE version 1.0 big-endian	0x5BE0
SBE version 1.0 little-endian	0xEB50

The Simple Open Framing Header specification also lists values for other wire formats.

SBE Message Encoding Header

The purpose of the message encoding header is to tell which message template was used to encode the message and to give information about the size of the message body to aid in decoding, even when a message template has been extended in a later version. See section 5 below for an explanation of the schema extension mechanism.

The fields of the SBE message header are:

- **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.
- **Template ID** - identifier of the message template
- **Schema ID** - identifier of the message schema that contains the template
- **Schema version** - the version of the message schema in which the message is defined

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. 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 4.3.1.

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="schemaId" primitiveType="uint16"/>
    <type name="version" primitiveType="uint16"/>
</composite>
```

The recommended header encoding is 8 octets.

Element	Description	Primitive type	Length (octets)	Offset
blockLength	Root block length	uint16	2	0

templateId	Template ID	uint16	2	2
schemaId	Schema ID	uint16	2	4
version	Schema Version	uint16	2	6

Optionally, implementations may support any other unsigned integer types for blockLength.

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 2.7.3 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.

Template ID

The identifier of a message type in a message schema. See section 4.5.2 below for schema attributes of a message.

Schema ID

The identifier of a message schema. See section 4.3.1 below for schema attributes.

Schema version

The version number of the message schema that was used to encode a message. See section 4.3.1 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. Position of a given field within any entry is fixed, with the exception of variable-length fields.

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>
```

Group block length

The blockLength part of a group dimension represents total space reserved for each group entry, not counting any nested repeating groups or variable-length fields. (Length of a variable-length Data field is given by its corresponding Length field.) Block length only represents message body fields; it does not include the length of the group dimension itself, which is a fixed size.

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.

Note that padding will only result in deterministic alignment if the repeating group contains no variable-length fields.

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 "Encoding of repeating group dimensions" section 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.

Empty group means nested group is empty

If a group contains nested repeating groups, then a NumInGroup counter of zero implies that both that group and its child groups are empty. In that case, no NumInGroup is encoded on the wire for the child groups.

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.

By default, the minimum number of entries is zero, and the maximum number is the largest value of the primitiveType of the counter.

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

The number of entries may be restricted to a specific range; see "Restricting repeating group entries" below.

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 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
```

Restricting repeating group entries

The occurrences of a repeating group may be restricted to a specific range by modifying the numInGroup member of the group dimension encoding. The minValue attribute controls the minimum number of entries, overriding the default of zero, and the maxValue attribute restricts the maximum entry count to something less than the maximum corresponding to its primitiveType. Either or both attributes may be specified.

Example of a restricted group encoding

```
<composite name="restrictedGroupSizeEncoding">
    <type name="blockLength" primitiveType="uint16"/>
    <type name="numInGroup" primitiveType="uint16" semanticType="NumInGroup"
minValue="1" maxValue="10" />
</composite>
```

Sequence of message body elements

Root level 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::

a. Fixed-length scalar fields, such as integers

b. Fixed-length character arrays

c. 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 elements

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 4.8 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.
Fixed-length field after repeating group or variable-length field	All fixed-length fields in the root of a message or in a repeating group entry must be listed before any (nested) repeating group or variable-length field.
Repeating group after variable-length field	All repeating groups at the root level or in a nested repeating group must be listed before any variable length field at the same level.

Message Schema

XML schema for SBE message schemas

See SimpleBinary1-0.xsd for the normative XML Schema Definition (XSD) for SBE.

XML namespace

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

`xmlns:sbe=http://fixprotocol.io/sbe/rc4`

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

Caution: Users should treat the SBE XML namespace as a URI (unique identifier), not as a URL (physical resource locator). Firms should not depend on access to the FIX Trading Community web site to validate XML schemas at run-time

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.

Capitalization

The value of a field's `semanticType` attribute is a FIX data type. In this document, FIX types are capitalized exactly as in the FIX repository, from which all official FIX documentation and references are derived. Since the capitalization is somewhat inconsistent, however, it is recommended that matching of type names should be case insensitive in schema parsers.

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	Should be unique between counterparties but no naming convention is imposed.
id	Unique identifier of a	unsignedInt		Should be unique

	schema			between counterparties
version	Version of this schema	nonnegativeInteger		Initial version is zero and is incremented for each version
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	An encoding with this name must be contained by '`.

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.

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 any sequence:

- Element `<type>` defines a simple encoding

- Element `<composite>` defines a composite encoding
- Element `<enum>` defines an enumeration
- 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.

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	Override of 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.
offset	If a member of a composite type, tells the offset from the beginning of the composite. By default, the offset is the sum of preceding element sizes, but it may be increased to effect byte alignment.	unsignedInt	optional	See section 4.4.4.3 below
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	Must be less than or equal to the version of the message schema.
deprecated	Documents the version of a schema in which a type was deprecated. It should no longer be used in	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.

new messages.

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.

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

<composite>

attribute	Description	XML type	Usage	Valid values
name	Name of encoding	symbolicName_t	required	Must be unique among all encoding types.
offset	The offset from the beginning of the composite. By default, the offset is the sum of preceding element sizes, but it may be increased to effect byte alignment.	unsignedInt	optional	
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	Must be less than or equal to the version of the message schema.

deprecated	Documents the version of a schema in which a type was deprecated. It should no longer be used in new messages.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.

Composite type elements

A `<composite>` composite encoding element may be composed of any combination of types, including `<type>` simple encoding, `<enum>` enumeration, `<set>` bitset, and nested composite type. The elements that compose a composite type carry the same XML attributes as stand-alone types.

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>
```

Element offset within a composite type

If a message designer wishes to control byte boundary alignment or map to an existing data structure, element offset may optionally be specified on a simple type, enum or bitset within a composite type. Offset is the number of octets from the start of the composite; it is a zero-based index.

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

Null value of a composite type

For a composite type, nullness is indicated by the value of its first element. For example, if a price field is optional, a null value in its mantissa element indicates that the price is null.

Reference to reusable types

A composite type often has its elements defined in-line within the `<composite>` XML element as shown in the example above. Alternatively, a common type may be defined once on its own, and then referred to by name with the composite type using a `<ref>` element.

<ref> attributes

<ref>

attribute	Description	XML type	Usage	Valid values
name	Usage of the type in this composite	symbolicName_t	required	
type	Name of referenced encoding	symbolicName_t	required	Must match a defined type, enum or set or composite name attribute.
offset	The offset from the beginning of the composite. By default, the offset is the sum of preceding element sizes, but it may be increased to effect byte alignment.	unsignedInt	optional	
sinceVersion	Documents the version of a schema in which a type was added	nonnegativeInteger	default = 0	Must be less than or equal to the version of the message schema.
deprecated	Documents the version of a schema in which a type was deprecated. It should no longer be used in new messages.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.

Type reference examples

Reference to an enum

In this example, a futuresPrice is encoded as 64 bit integer mantissa, 8 bit exponent, and a reused enum type.

```
<enum name="booleanEnum" encodingType="uint8" semanticType="Boolean">
    <validValue name="false">0</validValue>
    <validValue name="true">1</validValue>
</enum>

<composite name="futuresPrice">
    <type name="mantissa" primitiveType="int64" />
    <type name="exponent" primitiveType="int8" />
    <ref name="isSettlement" type="boolEnum" />
</composite>
```

Reference to a composite type

In this example, a nested composite is formed by using a reference to another composite type. It supports the expresson of a monetary amount with its currency, such as USD150.45. Note that a reference may carry an offset within the composite encoding that contains it.

```
<composite name="price">
    <type name="mantissa" primitiveType="int64" />
    <type name="exponent" primitiveType="int8" />
</composite>

<composite name="money">
    <type name="currencyCode" primitiveType="char" length="3"
semanticType="Currency" />
    <ref name="amount" type="price" semanticType="Price" offset="3" />
</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 *or* a primitive type: char uint8 uint16 uint32 uint64
sinceVersion	Documents the version	nonnegativeInteger	default =	Must be less

	of a schema in which a type was added		0	than or equal to the version of the message schema.
deprecated	Documents the version of a schema in which a type was deprecated. It should no longer be used in new messages.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.
offset	If a member of a composite type, tells the offset from the beginning of the composite. By default, the offset is the sum of preceding element sizes, but it may be increased to effect byte alignment.	unsignedInt	optional	

`<validValue>` element attributes

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

`<validValue>` 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	
deprecated	Documents the version of a schema in which a value was deprecated. It should no longer be used in new messages.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.

<validValue> element content

The element is required to carry a value, which is the valid value as a string. The string value in XML must be convertible to the data type of the encoding, such as an integer.

<enum> and <validValue> elements

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 *or* a primitive type: uint8 uint16 uint32 uint64
sinceVersion	Documents the version of a schema in which a type was added	nonnegativeInteger	default = 0	Must be less than or equal to the version of the message schema.
deprecated	Documents the version of a schema in which a type was deprecated. It should no longer be used in new messages.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.
offset	If a member of a composite type, tells the offset from the beginning of the composite. By default, the offset is the sum of preceding element sizes, but it may be increased to effect byte alignment.	unsignedInt	optional	

`<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	
deprecated	Documents the version of a schema in which a choice was deprecated. It	nonnegativeInteger	optional	Must be less than or equal to the version

should no longer be used in new messages.

of the message schema.

< choice > element content

The element is required to carry a value, which is an unsigned integer representing a zero-based index to a bit within a bitset. Zero is the least significant bit.

<set> and <choice> XML elements

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	Must be unique among all messages in a schema
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	token	optional	Listed in FIX specifications
sinceVersion	Documents the version of a schema in which a message was added	nonNegativeInteger	default = 0	
deprecated	Documents the version of a schema in which a message was deprecated. It should no longer be sent but is documented for back-compatibility.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.

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.

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, `<enum>` or `<set>`.
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. constant = constant value not sent on wire.
valueRef	Constant value of a field as a valid value of an	qualifiedName_t	optional Valid only if	If provided, the qualified name must match the

	enumeration		presence= ”constant”	name attribute of a <validValue> within an <enum>
sinceVersion	The version of a message schema in which this field was added.	InonnegativeInteger	default=0	Must not be greater than version attribute of <messageSchema> element.
deprecated	Documents the version of a schema in which a field was deprecated. It should no longer be used in new messages.	nonnegativeInteger	optional	Must be less than or equal to the version of the message schema.

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 since they both inherit attributes from the blockType XML type. A group has the same child members as a message, and they must appear in the same order:

1. Element <field> defines a fixed-length field
2. Element <group> defines a repeating group. Groups may be nested to any level.
3. Element <data> defines a variable-length field, such as raw data

The number of members of each type is unbound.

<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.

<group> element inherits attributes of blockType. See <message> above.

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.
presence 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.

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	timestamp, TZ time
timezoneHour	TZ time
timezoneMinute	TZ time
unit	timestamp, 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.

This specification only details compatibility at the presentation layer. It does not relieve application developers of any responsibility for carefully planning a migration strategy and for handling exceptions at the application layer.

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 and if there are no subsequent variable data elements at the end of the message.
- A variable data element may be added, but only after existing groups and data.
- Message header encoding cannot change.
- In general, metadata changes such as name or description corrections do not break compatibility so long as wire format does not change.

Changes that break those constraints require consumers to update to the current schema used by publishers. An message template that has changed in an incompatible way must be assinged a new template "id" attribute.

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 4.3.1 above for schema attributes.

Since version

When a new field, enumeration value, 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 the root level of the message may optionally be documented on a `<message>` element in the schema using the blockLength attribute. See section 4.5.3 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 3.4.8.2 above.

Deprecated elements

A message schema may document obsolete elements, such as messages, fields, and valid values of enumerations with deprecated attribute. Updated applications should not publish deprecated messages or values, but declarations may remain in the message schema during a staged migration to replacement message layouts.

Wire format features for extension

Message size

It is assumed that a either message boundaries are delimited by a transport or 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 3.2.2 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.

Comaptibility strategy

This suggested strategy is non-normative.

A message decoder compares the schema version in a received message header to the version that the decoder was built with.

If the *received version is equal to the decoder's version*, then all fields known to the decoder may be parsed, and no further analysis is required.

If the *received version is greater than the decoder's version* (that is, the producer's encoder is newer than the consumer's decoder), then all fields known to the decoder may be parsed but it will be unable to parse added fields.

Also, an old decoder may encounter unexpected enumeration values. The application layer determines whether an unexpected value is a fatal error. Probably so for a required field since the business meaning is unknown, but it may choose to allow an unknown value of an optional field to pass through. For example, if OrdType value J="Market If Touched" is added to a schema, and the consumer does not recognize it, then the application returns an order rejection with reason "order type not supported", even if it does not know what "J" represents. Note that this is not strictly a versioning problem, however. This exception handling is indistinguishable from the case where "J" was never added to the enum but was simply sent in error.

If the *received version is less than the decoder's version* (that is, the producer's encoder is older than the consumer's decoder), then only the fields of the older version may be parsed. This information is available through metadata as "sinceVersion" attribute of a field. If sinceVersion is greater than received schema version, then the field is not available. How a decoder signals an application that a field is unavailable is an implementation detail. One strategy is for an application to provide a default value for unavailable fields.

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

The example messages are preceded by Simple Open Framing Header. Note that SOFH encoding is always big-endian, regardless of the byte order of the SBE message body. See that FIX standard for details.

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

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>`

<sbe:messageSchema
    xmlns:sbe="http://fixprotocol.io/sbe/rc4"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    package="Examples" id="100"
    description="Test dictionary"
```

```
    byteOrder="littleEndian"
    xsi:schemaLocation="http://fixprotocol.io/sbe/rc4 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">
        <type name="blockLength" primitiveType="uint16"/>
        <type name="templateId" primitiveType="uint16"/>
        <type name="schemaId" primitiveType="uint16"/>
        <type name="version" primitiveType="uint16"/>
    </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 constrained 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):

00 00 00 44 eb 50 36 00 63 00 64 00 00 00 4f 52 : D P6 c d OR

44 30 30 30 30 31 41 43 43 54 30 31 00 00 47 45 :D00001ACCT01 GE

4d 34 00 00 00 00 31 00 84 68 90 fe a8 9a 13 07 :M4 1 h

00 00 00 32 1a 85 01 00 00 00 00 00 00 00 00 00 : 2

00 00 00 80

Interpretation

Wire format	Field ID	Name	Offset	Length	Interpreted value
00000044		Simple Open Framing Header		4	Message size=68
eb50		Simple Open Framing Header		2	SBE version 1.0 little-endian

Wire format	Field ID	Name	Offset	Length	Interpreted value
3600		messageHeader blockLength		2	Root block size=54
6300		messageHeader templateId		2	Template ID=99
6400		messageHeader schemaId		2	Schema ID=100
0000		messageHeader version		2	Schema version=0
4f52443030303031	11	ClOrdID	0	8	ORD00001
4143435430310000	1	Account	8	8	ACCT01
47454d3400000000	55	Symbol	16	8	GEM4
31	54	Side	24	1	1 Buy
c021ed1b04c32b13	60	TransactTime	25	8	2013-10-10 13:35:33.135 as nanoseconds since UNIX

Wire format	Field ID	Name	Offset	Length	Interpreted value
					epoch
07000000	38	OrderQty	33	4	7
32	40	OrdType	37	1	2 Limit
1a85010000000000	44	Price	38	8	99.610
0000000000000008	99	StopPx	46	8	null

Message with a repeating group

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

Sample execution report message schema

Add this encoding types element 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="ExecutionReport" 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 execution message

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

```
00 00 00 54 eb 50 2a 00 62 00 64 00 00 00 4f 30 : T P* b d O0
30 30 30 30 30 31 45 58 45 43 30 30 30 30 46 31 :000001EXEC0000F1
47 45 4d 34 00 00 00 00 de 07 06 ff ff 31 01 00 :GEM4 1
00 00 06 00 00 00 dd 3f 0c 00 02 00 1a 85 01 00 : ?
00 00 00 00 02 00 00 00 24 85 01 00 00 00 00 00 : $
04 00 00 00
```

Interpretation

Offset is from beginning of block.

Wire format	Field ID	Name	Offset	Length	Interpreted value
00000054		Simple Open Framing Header		4	Message size=84
eb50		Simple Open Framing Header		2	SBE version 1.0 little-endian
2a00		messageHeader blockLength		2	Root block size=42
6200		messageHeader templateId		2	Template ID=98
6400		messageHeader		2	Schema ID=100

		schemaId			
0000		messageHeader version		2	Schema version=0
4f30303030303031	37	OrderID	0	8	O0000001
4558454330303030	17	ExecID	8	8	EXEC0000
46	150	ExecType	16	1	F Trade
31	39	OrdStatus	17	1	1 PartialFilled
47454d3400000000	55	Symbol	18	8	GEM4
de0706ffff	200	MaturityMonthYear	26	5	201406
31	54	Side	31	1	1 Buy
01000000	151	LeavesQty	32	4	1
06000000	14	CumQty	36	4	6
753e	75	TradeDate	40	2	2013-10-11
0c00	2112	groupSizeEncoding			FillsGrp block size=12
0200	1362	groupSizeEncoding			FillsGrp NumInGroup=2
1a85010000000000	1364	FillPx	0	8	FillsGrp instance 0
02000000	1365	FillQty	8	4	2
2485010000000000	1364	FillPx	0	8	FillsGrp instance 1
04000000	1365	FillQty	8	4	4

Message with a variable-length field

Sample business reject message schema

Add this encoding types element to those in the previous example.

```
<types>
    <type name="intEnumEncoding" primitiveType="uint8"/>

    <composite name="DATA" description="Variable-length data">
        <type name="length" primitiveType="uint16" />
        <type name="varData" length="0" primitiveType="uint8">
    </composite>

    <enum name="businessRejectReasonEnum" encodingType="intEnumEncoding">>
        <validValue name="Other">0</validValue>
        <validValue name="UnknownID">1</validValue>
        <validValue name="UnknownSecurity" >2</validValue>
        <validValue name="ApplicationNotAvailable" >4</validValue>
        <validValue name="NotAuthorized" >6</validValue>
```

```
    </enum>

</types>

    <sbe:message name="BusinessMessageReject" id="97"
        blockLength="9" semanticType="j">
        <field name="BusinesRejectRefId" id="379" type="idString"
            offset="0" semanticType="String" />
        <field name="BusinessRejectReason" id="380"
type="businessRejectReasonEnum"
            offset="8" semanticType="int" />
        <data name="Text" id="58" type="DATA" semanticType="data" />
    </sbe:message>
```

Wire format of a business reject message

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

00 00 00 40 eb 50 09 00 61 00 64 00 00 00 4f 52 : @ P a d OR

44 30 30 30 30 31 06 27 00 4e 6f 74 20 61 75 74 :D00001 ' Not aut

68 6f 72 69 7a 65 64 20 74 6f 20 74 72 61 64 65 :horized to trade

20 74 68 61 74 20 69 6e 73 74 72 75 6d 65 6e 74 : that instrument

Interpretation

Wire format	Field ID	Name	Offset	Length	Interpreted value
00000040		Simple Open Framing Header		4	Message size=64
eb50		Simple Open Framing Header		2	SBE version 1.0 little-endian
0900		messageHeader blockLength		2	Root block size=9
6100		messageHeader templateId		2	Template ID=100
6400		messageHeader		2	Schema ID=0

Wire format	Field ID	Name	Offset	Length	Interpreted value
		schemaId			
0000		messageHeader version		2	Schema version=0
4f52443030303031	379	BusinessRejectRefId	0	8	ORD00001
06	380	BusinessRejectReason	8	1	6 NotAuthorized
2700		DATA length		2	length=39
4e6f74206175746 8 6f72697a656420 74 6f207472616465 20 7468617420696e 73 7472756d656e74		DATA varData			39 Not authorized to trade that instrument

Release Notes

Release Candidate 4

These issues were resolved and accepted for Release Candidate 4. See issues and pull requests
in GitHub for details and changes.

Issue	Description	Section
2	Schema extension is vague in terms of what compatibility means	5
3	Extensibility of the Template ID	5
6	Limiting maximum occurences of repeating group	3
8	blockLength for repeating groups of variable length	3
11	Offsets within composite types	4
12	Composites reusing other types	4

Release Candidate 3

This is a summary of document changes to Release Candidate 3 from RC2. Changes in this release were intended only as clarifications or to add capabilities. Message schemas that conformed to the RC2 specification should still conform to the RC3 wire format.

Section 1

References section expanded.

Section 2

- Statement added that non-FIX data types should not carry a semanticType attribute in a message schema.
- String encoding section split into two sections for strings (text fields) and data (non-character data) to clarify the distinction. Both text and non-text can be either fixed-length `<field>` or variable-length `<data>`.
- Timestamp encoding enhanced to allow time unit to either be specified as a constant in a message schema or to be serialized on the wire.

Section 3

Message structure is enhanced to allow variable-length `<data>` elements within a repeating group entry.

Section 4

Message schema XSD updated to support `<data>` in repeating groups and for various other refinements

Section 5

- Statements added to say whole repeating groups or variable data may be added to a message without breaking compatibility so long as the added elements are at the end of a message.
- Added deprecated schema attribute to mark obsolete elements.

Section 6

No change

Section 7

Examples updated to use Simple Open Framing Header.

Section 8

Release notes added.