FIX PROTOCOL
INDUSTRY-DRIVEN MESSAGING STANDARD℠

FIX Simple Binary Encoding v0.9.2

RELEASE CANDIDATE 1: MAY 28, 2013

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

High Performance Working Group
February 2013

Contents

Document History

Revision	Date	Author	Revision comments
0.9	Jan. 31, 2013	Don Mendelson CME Group	Submitted to High Performance Working Group
0.9.1	Feb. 6, 2013	Don Mendelson CME Group	Added “core feature” subtitle, added “may” to documentation language, corrected int64 range values, changed primitive types like “uint64” to lower case, added optional integer types to decimal encoding, changed description of NumInGroup field position, added usage guideline section with identifier encodings.
0.9.2	Feb. 26, 2013	Don Mendelson CME Group	Subtitle removed (document to remain as one volume), recommended minimal message header explained, sequence of message body elements updated to cover the case of nested repeating groups, message structure validations added.
0.9.2 RC1	May 28, 2013		Promoted to Release Candidate 1 for public release.

1 Introduction

FIX Binary message encoding 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.

1.1 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.

1.2 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.

1.3 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.

1.4 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 the components of a message 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 template that conforms to an XML schema that is published as part of this standard. Also known as a message template or metadata.

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.

1.5 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.

1.5.1 Specification terms

These key words in this document are to be interpreted as described in Internet Engineering Task Force RFC 2119. These terms indicate an absolute requirement for implementations of the standard: "**must**", or "**required**".

This term indicates an absolute prohibition: "**must not**".

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

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

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

1.5.2 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" fixUsage="int" />
```

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

1.6 10270000 References

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

XML 1.1 schema standards are located here W3C XML Schema

2 Field Encoding

2.1 Field aspects

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

2.1.1 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 `fixUsage`. See the section 2.2 below for a listing of those FIX types.

2.1.2 Encoding

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

2.1.3 Metadata

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

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

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

See section 4 below for a detailed message schema specification.

2.1.4 Field presence

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

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

2.2 FIX data type summary

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

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

FIX semantic type	Binary type	Section	Description
int	Integer encoding	2.4	An integer number
Length	Integer encoding	2.4	Field length in octets. Value must be non-negative.
TagNum	Integer encoding	2.4	A field's tag number. Value must be positive.
SeqNum	Integer encoding	2.4	A field representing a message sequence number. Value must be positive
NumInGroup	NumInGroup encoding	2.4	A field 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	Decimal encoding	2.5	A fractional number
Price	Decimal encoding	2.5	A decimal number representing a price
PriceOffset	Decimal encoding	2.5	A decimal number representing a price offset, which can be mathematically added to a Price.
Amt	Decimal encoding	2.5	A field typically representing a Price times a Qty.
Percentage	Decimal encoding	2.5	A field representing a percentage (e.g. 0.05 represents 5% and 0.9525 represents 95.25%).
char	Character	2.6.1	Single ASCII character value. Can include any alphanumeric character or punctuation. All char fields are case sensitive (i.e. m != M).

FIX semantic type	Binary type	Section	Description
String	Fixed size character array	2.6.2	A fixed-length character array of ASCII encoding
String	Variable length string	2.6.3	Alpha-numeric free format strings can include any character or punctuation. All String fields are case sensitive (i.e. morstatt != Morstatt). ASCII encoding.
String—EncodedText	String encoding	2.6.3	Non-ASCII string. The character encoding may be specified by a schema attribute.
data	String encoding	2.6.3	Variable-length data. Must be paired with a Length field.
XMLData	String encoding	2.6.3	Variable-length XML. Must be paired with a Length field.
Country	Fixed length character array; size = 2 or a subset of values may use Enumeration encoding	2.6.2	ISO 3166 Country code
Currency	Fixed length character array; size = 3 or a subset of values may use Enumeration encoding	2.6.2	ISO 4217 Currency code (3 character)
Exchange	Fixed length character array; size = 4 or a subset of values may use Enumeration encoding	2.6.2	ISO 10383 Market Identifier Code (MIC)
Language	Fixed length character array; size = 2 or a subset of values may use Enumeration encoding	2.6.2	National language - uses ISO 639-1 standard
Implicit enumeration—char or int	Enumeration encoding	2.11	A single choice of alternative values
Boolean	Boolean encoding	2.11.5	Values true or false

FIX semantic type	Binary type	Section	Description
MultipleCharValue	Multi-value choice encoding	2.12	Multiple choice of a set of values
MultipleStringValue	Multi-value choice encoding**.** String choices must be mapped to int values.	2.12	Multiple choice of a set of values
MonthYear	MonthYear encoding	2.7	A flexible date format that must include month and year at least, but may also include day or week.
UTCTimestamp	Date and time encoding	2.8	Time/date combination represented in UTC (Universal Time Coordinated, also known as "GMT")
UTCTimeOnly	Date and time encoding	2.8	Time-only represented in UTC (Universal Time Coordinated, also known as "GMT")
UTCDateOnly	Date and time encoding	2.8	Date represented in UTC (Universal Time Coordinated, also known as "GMT")
LocalMktDate	Local date encoding	2.8	Local date(as oppose to UTC)
TZTimeOnly	TZTimeOnly	2.10.3	Time of day
TZTimestamp	TZTimestamp	2.10.1	Tme/date combination representing local time with an offset to UTC to allow identification of local time and timezone offset of that time. The representation is based on ISO 8601

2.3 Common field schema attributes

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

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

2.3.1 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.

2.4 Integer encoding

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

2.4.1 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

2.4.2 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.

Type: Schema attribute	`Int8`	`uint8`	`Int16`	`uint16`	`int32`	`uint32`	`int64`	`uint64`
`minValue`	–127	0	–32767	0	$-2^{31}+1$	0	$-2^{63}+1$	0
`maxValue`	127	254	32767	65534	$2^{31}-1$	$2^{32}-2$	$-2^{63}-1$	$2^{64}-2$
`nullValue`	–128	255	–32768	65535	-2^{31}	$2^{32}-1$	-2^{63}	$2^{64}-1$

2.4.3 Byte order

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

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

2.4.4 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" />
```

2.4.5 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" fixUsage="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

```
<field type="uint8" name="MaxPriceLevels" id="1090"
   fixUsage="int" maxValue=”6” presence=”optional”
   nullValue=”255” />
```

Wire format of uint8 value decimal 3.

```
03
```

Sequence number field with integer encoding

```
<field type="uint64" name="MsgSeqNum" id="34"
   fixUsage="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 $2^{32} - 1$

```
ffffffff
```

2.5 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.

2.5.1 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.

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

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

Encoding type	Description	Backing primitives	Length (octets)
`decimal`	Floating-point decimal	Composite: Int64 mantissa Int8 exponent	9
`decimal64`	Fixed-point decimal	Int64 mantissa constant exponent	8
`decimal32`	Fixed-point decimal	Int32 mantissa constant exponent	4

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

2.5.2 Range attributes for decimal fields

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

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

Type: Schema attribute	`decimal`	`decimal64`	`decimal32`
exponent range	Valid range –128 to 127	Valid range –128 to 127	Valid range –128 to 127
`minValue`	-10^{-128}	-10^{-128}	-10^{-128}

Type: Schema attribute	decimal	decimal64	decimal32
maxValue	$2^{64} * 10^{127}$	$2^{32} * 10^{127}$	$2^{64} * 10^{127}$
nullValue	mantissa=-2^{64}	-2^{32}	-2^{64}

2.5.3 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.

Recommended decimal encoding specifications

```
<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>
```

2.5.4 Composite encoding packing

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

2.5.5 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" fixUsage="Qty">
   <type name="mantissa" primitiveType="int32" />
   <type name="exponent" primitiveType="int8"
   presence="constant">0</type>
</composite>
```

Field inherits `fixUsage` 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`

2.6 String encodings

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

2.6.1 Character

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

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

2.6.1.1 Range attributes for char fields

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

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

2.6.1.2 Encoding of char type

This is the standard encoding for `char` type.

```
<type name="char" primitiveType="char" fixUsage="char" />

Wire format of char encoding of "A" (ASCII value 65, hexadecimal 41)
41
```

2.6.2 Fixed-length character array

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

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

2.6.2.1 Encoding specifications for fixed-length character array

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

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

2.6.2.2 Examples of fixed-length character arrays

```
A typical string encoding specification
<type name="string6" primitiveType="char" fixUsage="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="MsgTypeE" fixUsage="String"
   primitiveType="char" length="2" presence="constant">E
</type>

<field type="MsgTypeE" name="MsgType" id="35" />
```

2.6.3 Variable-length data encoding

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

FIX data type	Description	Backing primitives	Length (octets)
`Length`	The length of variable data in octets	`primitiveType=uint8 or uint16`	1 or 2

`data`	Raw data	Array of char of size specified in associated `Length` field. The data field itself should be specified as variable length.
		`primitiveType="char"`
		`variableLength="true"`
			variable

2.6.4 Range attributes for variable-length data fields

Key

Y=required
N=not required
X=not allowed

Schema attribute	`length`	`data`
`minValue`	0	X
`maxValue`	65534	X
`nullValue`	65535	X

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

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

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

2.6.5 Encoding specifications for variable-length data

Encoding specification for optional length up to 254 octets

```
<type name="length8" primitiveType="uint8" maxValue="254"
   presence="optional" nullValue="255" fixUsage="Length" />
```

Encoding specification for optional length up to 65534 octets

```
<type name="length16" primitiveType="uint16" maxValue="65534"
   presence="optional" nullValue="65535" fixUsage="Length" />
```

The standard encoding for data

```
<type name="rawData" primitiveType="char" variableLength="true"
fixUsage="data" />
```

2.6.6 Example of data fields

Examples show encoded bytes on the wire.

Wire format of `Length` as `uint8`, in hexadecimal format

```
04
```

Wire format of `data` in character and hexadecimal formats

```
M  S  F  T
4d534654
```

2.7 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

2.7.1 Composite encoding packing

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

2.7.2 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" fixUsage="MonthYear">
   <type name="year" primitiveType="uint16" presence="optional"
   nullValue="65536" />
   <type name="month" primitiveType="uint8" minValue="1"
   maxValue="12" />
   <type name="day" primitiveType="uint8" minValue="1"
   maxValue="31" presence="optional" nullValue="255" />
   <type name="week" description="week of month"
   primitiveType="uint8" minValue="1" maxValue="5"
   presence="optional" nullValue="255" />
</composite>
```

Example MonthYear field specification

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

Wire format of MonthYear 2014 June week 3 as hexadecimal

```
de0706ff03
```

2.8 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).

2.8.1 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.

2.8.2 Time unit

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

FIX data type	Description	Backing primitives	Length (octets)	Schema attributes

`UTCTimestamp`	UTC date/time
	Default: nanoseconds since Unix epoch Range Jan. 1, 1970 - July 21, 2554
		`uint64`	8	`epoch=”unix” (default)`
				
				`timeUnit = second | millisecond | microsecond | nanosecond (default)`

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

2.8.3 Examples of date/time fields

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

```
<type name="timestamp" primitiveType="uint64"
   timeUnit="nanosecond" fixUsage="UTCTimestamp" />

Wire format of UTCTimestamp
007420bf8838fb17
```

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

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

Wire format of UTCTimeOnly
42dc561600000000
```

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

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

Wire format of UTCDateOnly

```
204e
```

2.9 Local date encoding

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

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

The standard encoding specification for LocalMktDate

```
<type name="localMktDate" primitiveType="uint16"
   fixUsage="LocalMktDate" />
```

2.10 Local time encoding

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

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

2.10.1 TZTimestamp encoding

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

FIX data type	Description	Backing primitives	Length (octets)	Schema attributes

`TZTimestamp`	date/time with timezone	`uint64`
		Default: nanoseconds since Unix epoch Range Jan. 1, 1970 - July 21, 2554
			8	`epoch="unix" (default)`
				`Represents` Jan. 1, 1970 local time
				
				`timeUnit = second | millisecond | microsecond | nanosecond (default)`

FIX data type	Description	Backing primitives	Length (octets)	Schema attributes
	Time zone hour offset	`Int8`	1	`None`
	Time zone minute offset	`uint8`	1	`None`

2.10.2 Composite encoding packing

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

Standard TZTimestamp encoding specification

```
<composite name="tzTimestamp" fixUsage="TZTimestamp">
   <type name="time" primitiveType="uint64"
   timeUnit="nanosecond" />
   <!-- Sign of timezone offset is on hour subfield -->
   <type name="timezoneHour" primitiveType="int8"
   minValue="-12" maxValue="14" />
   <type name="timezoneMinute" primitiveType="uint8"
   maxValue="59"  />
</composite>
```

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

`0050d489fea22413fa00`

2.10.3 TZTimeOnly encoding

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

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

FIX data type	Description	Backing primitives	Length (octets)	Schema attributes
`TZTimeOnly`	Time of day only with time zone	`uint64` Default: nanoseconds since midnight today, local time	8	`timeUnit = second | millisecond | microsecond | nanosecond (default)`
	Time zone hour offset	`Int8`	1	`None`

FIX data type	Description	Backing primitives	Length (octets)	Schema attributes
	Time zone minute offset	`uint8`	1	None

2.10.4 Composite encoding packing

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

Standard TZTimeOnly encoding specification

```
<composite name="tzTimeOnly" fixUsage="TZTimeOnly">
   <type name="time" primitiveType="uint64"
   timeUnit="nanosecond" />
   <!-- Sign of timezone offset is on hour subfield -->
   <type name="timezoneHour" primitiveType="int8"
   minValue="-12" maxValue="14" />
   <type name="timezoneMinute" primitiveType="uint8"
   minValue="0" maxValue="59"  />
</composite>
```

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

`006c5ebe76000000fa00`

2.11 Enumeration encoding

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

2.11.1 Primitive type encodings

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

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

2.11.2 Value encoding

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

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

2.11.3 Enumeration of values

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

2.11.4 Enumeration examples

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

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

```
<enum SideEnum="Side" encodingType="char">
   <validValue name="Buy">1</validValue>
   <validValue name="Sell">2</validValue>
   <validValue name="SellShort">5</validValue>
   <validValue name="SellShortExempt">6</validValue>
   <!-- not all FIX values shown -->
</enum>
```

Side field specification references the enumeration

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

Wire format of Side "Buy" code as hexadecimal

```
31
```

Example of a char field using a constant enum value

```
<type name="GeneralIdentifier" primitiveType="char"
   fixUsage="char" description="source of the PartyID"
   presence="constant">C</type>

<field type="GeneralIdentifier" name="PartyIDSource" id="447"
   description="Party ID source is fixed"/>
```

2.11.5 Boolean encoding

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

Standard encoding specifications for required and optional Boolean fields

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

<enum name="boolean" encodingType="uint8" fixUsage="Boolean">
   <validValue name="false">0</validValue>
   <validValue name="true">1</validValue>
</enum>

<enum name="optionalBoolean" encodingType="uint8"
   presence="optional" nullValue="255" fixUsage="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
```

2.12 Multi-value choice encoding

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

2.12.1 Primitive type encodings

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

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

2.12.2 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.

2.12.3 Enumeration of values

In a message schema, the choices are specified as `<validValue>` members of an `<enum>`. Choices are assigned values as an ordinal of bits in the bit set. The first choice is assigned the least significant bit, and so forth.

2.12.4 Multi-value example

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

```
<enum name="FinancialStatusEnum" encodingType="uint8"">
   <validValue name="Bankrupt">1</validValue>
   <validValue name="Pending delisting">2</validValue>
   <validValue name= Restricted">3</validValue>
```

```
</enum>

<field type="FinancialStatus" name="FinancialStatusEnum"
   id="291" fixUsage="MultipleCharValue"/>
```

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

```
03
```

2.13 NumInGroup encoding

A `NumInGroup` field represents the number of entries in a repeating group. For a description of repeating groups, see section 3.3 below.

2.13.1 Primitive type encodings

In most cases, an 8 bit integer will be sufficient to hold the number of entries in a repeating group, but a larger unsigned data type may be specified if necessary.

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

2.13.2 Empty repeating group

A `NumInGroup` value of zero means that no entries are encoded in that particular message. In that case, the `NumInGroup` field is still transmitted, but no space is reserved for the repeating group that it is associated with.

2.13.3 Encoding specifications

The required schema attribute `groupName` associates a `NumInGroup` field to its repeating group by name.

2.13.4 NumInGroup example

A `NumInGroup` encoding specification to handle up to 255 entries per group

```
<type name="numInGroup" primitiveType="uint8"
   fixUsage="NumInGroup" />

<field type="numInGroup" name="NoOrders" id="73"
   groupName="ListOrdGrp" />
<group name="ListOrdGrp ">
```

Wire format of `NumInGroup` 3 entries

```
03
```

2.14 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.
String contains invalid characters	A String contains non-ASCII printable characters.
Required subfields not populated in `MonthYear`	Year and month must be populated with non-null values, and the month must be in the range 1-12.
`UTCTimeOnly` exceeds day range	The value must not exceed the number of time units in a day, e.g. greater than 86400 seconds.
`TZTimestamp` and `TZTimeOnly` must carry a valid 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.

3 Message Structure

3.1 Message Header

Each message may be prefixed with a message header dictated by the session protocol in use. Simple Binary Encoding imposes two requirements for the message header:

1. Total message length in octets, including the header and message body.
2. Message template ID. That is, an identifier of the message schema used to encode the message.

For best performance, these two items should be the first two fields in the header and should be presented in that order. They may be followed by other fields required by the session protocol for message delivery, e.g. message sequence number. The session protocol specification defines the header size if it is a fixed length, or should provide some other means to determine an offset to the message body that follows.

3.1.1 Minimal header

Without requirements imposed by a session protocol for delivery and identification of messages, a minimalist message header may be encoded in Simple Binary Encoding with the sole purpose of enabling decoding.

The header fields precede the message body of every message in a fixed position.

Recommended minimal header

```
<field name="MsgSize" id="9999" type=" uint16"
   fixUsage="int"/>
<field name="TemplateId" id="9998" type="uint16"
   fixUsage="int"/>
```

Field	Size	Offset
Message size	2	0
Template ID	2	2

Tip

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

3.2 Message Body

The message body conveys the business information of the message.

3.2.1 Data only on the wire

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

3.2.2 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.

3.2.3 Field position and padding

3.2.3.1 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.

Example of fields with default positions

```
<field name="ClOrdId" id="11" type="string14"
   fixUsage="String"/>
<field name="Side" id="54" type="char" fixUsage="char"/>
<field name="OrderQty" id="38" type="intQty32" fixUsage="Qty"/>
<field name="Symbol" id="55" type="string8" fixUsage="String"/>
```

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

3.2.3.2 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"
   fixUsage="String"/>
<field name="Side" id="54" type="char" offset="14"
   fixUsage="char"/>
<field name="OrderQty" id="38" type="intQty32" offset="16"
   fixUsage="Qty"/>
<field name="Symbol" id="55" type="string8" offset="20"
   fixUsage="String"/>
```

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

3.2.3.3 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">
```

3.3 Repeating Groups

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

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

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

3.3.1 Schema specification of a group

A repeating group is defined in a message schema by adding a `<group>` element to a 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" blockLength="16">
   <field name="PartyID" id="448" type="string14"
   fixUsage="String"/>
   <field name="PartyIDSource" id="447" type="char"
   fixUsage="char"/>
   <field name="PartyRole" id="452" type="uint8"
   fixUsage="int"/>
</group>
```

3.3.2 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.

3.3.3 Entry count field

Each group is associated with a required counter field of data type `NumInGroup` to tell how many entries are contained by a message. The value of the counter field is a non-negative integer. See section 2.13 above for encoding of that field.

There must be one and only one `NumInGroup` field per repeating group. The counter field is associated with group by a match between its `groupName` attribute and the `name` attribute of the group.

Elements are listed in a message schema in the same order that they are encoded on the wire. A `NumInGroup` field must be listed before the repeating group itself.

Example count field and its repeating group specification

```
<field type="uint8" name="NoOrders" id="73"
   groupName="ListOrdGrp" fixUsage=”NumInGroup”/>
<group name="ListOrdGrp ">
```

3.3.4 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` field. If the counter field is set to zero, then no entries are sent in the message, and no space is reserved for entries. The counter field itself is still transmitted, however.

3.3.5 Multiple repeating groups

A message may contain multiple repeating groups at the same level. Message schemas are free to either place all `NumInGroup` fields prior to all repeating groups at the same level or to place each `NumInGroup` counter prior to its associated group. In either alternative, the wire format follows the order listed in the schema.

Example of encoding specification with multiple repeating groups

```
<field type="uint8" name="NoContraBrokerss" id="382"
   groupName="ContraGrp" fixUsage=”NumInGroup”/>
<field type="uint8" name="NoAllocs" id="78"
   groupName="PreAllocGrp" fixUsage=”NumInGroup”/>
<group name="ContraGrp ">[ContraGrp group fields]</group>
<group name="PreAllocGrp ">[PreAllocGrp group fields]</group>
```

Alternative encoding specification with multiple repeating groups

```
<field type="uint8" name="NoContraBrokerss" id="382"
   groupName="ContraGrp" fixUsage=”NumInGroup”/>
<group name="ContraGrp ">[ContraGrp group fields]</group>
<field type="uint8" name="NoAllocs" id="78"
   groupName="PreAllocGrp" fixUsage=”NumInGroup”/>
<group name="PreAllocGrp ">[PreAllocGrp group fields]</group>
```

3.3.6 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

```
<field name="NoOrders" id="73" type="uint8"
   groupName="ListOrdGrp" fixUsage="NumInGroup"/>
<group name="ListOrdGrp" blockLength="31">
   <field name="ClOrdID" id="11" type="string14"
   fixUsage="String"/>
   <field name="ListSeqNo" id="67" type="uint32"
   fixUsage="int"/>
   <field name="Symbol" id="55" type="string8"
   fixUsage="String"/>
   <field name="Side" id="54" type="char" fixUsage="char"/>
   <field name="OrderQty" id="38" type="intQty32"
   fixUsage="Qty"/>
   <field name="NoPartyIDs" id="453" type="uint8"
   groupName="Parties" fixUsage="NumInGroup"/>
   <group name="Parties" parentGroupName="ListOrdGrp"
      blockLength="16">
      <field name="PartyID" id="448" type="string14"
         fixUsage="String"/>
      <field name="PartyRole" id="452" type="int"
         fixUsage="int"/>
   </group>
</group>
```

3.3.7 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.

3.4 Sequence of message body elements

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

1. Fixed-length fields that reside at the root level of the message (that is, not members of repeating groups), including any of the following, in the order specified by the message schema:
    a. `NumInGroup` associated with top-level (not nested) repeating groups. Any `NumInGroup` field must be positioned before its associated repeating group, but the `NumInGroup` field may or may not be adjacent to its repeating group.
    b. `Length` fields associated with data fields (variable length). Any `Length` field must be positioned before its associated data field, but the `Length` field may or may not be adjacent to the data.

   c. Fixed-length character arrays
   d. Any other fixed-length fields, such as integers
2. Repeating group entries, if any. Like the root level of the message, a repeating group entry is divided into a fixed-length portion and optionally, a variable-length portion.
   a. The fixed-length portion of a repeating group entry may contain any of the following, in the order specified by the message schema:
      i. `NumInGroup` field associated with nested repeating groups
      ii. Fixed-length character arrays
      iii. Any other fixed-length fields, such as integers
   b. The variable-length portion would contain nested repeating group entries, if any. If there are no nested groups, then the repeating group entries are of fixed-length and direct access to their elements is supported. On the other hand, repeating groups may be nested recursively, but entry position is not deterministic in that case.
3. Data fields, including raw data and variable-length strings, if any. Since `Length` fields are fixed-length, they will *all* be positioned before any `data` fields, and a `Length` field may not be adjacent to its associated `data` field.

3.5 Message structure validation

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

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

Error condition	Error description
Wrong message size in header	A message size value smaller than the actual message may cause a message to be truncated.
Wrong message template ID in header	A mismatch of message schema would likely cause fields to be misinterpreted.
`NumInGroup` field missing for repeatinggroup or group missing for `NumInGroup`	There must be a one-to-one relationship between a repeating group and a counter field.
`NumInGroup field follows its associated repeating group`	A `NumInGroup` counter field must precede its associated repeating group.
`Length` field missing for data field or data missing for `Length`	There must be a one-to-one relationship between a `Length` field and a variable length data field.
`Length` field follows its associated data field	A `Length` field must precede its associated data field.

4 Message Schema

4.1 XML namespace

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

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

Conventionally, the URL is aliased by the prefix "sbe".

4.2 Root element

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

4.2.1 <messageSchema> attributes

The root element provides basic identification of a schema.

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

Schema attribute	Description	XML type	Usage	Valid values
package	Name or category of a schema	string	optional	
version	Version of FIX semantics	string	optional	
byteOrder	Byte order of encoding	token	default=littleEndian	littleEndian bigEndian
description	Documentation of the schema	string	optional	

<messageSchema> element

```
<xs:element name="messageSchema">
   <xs:annotation><xs:documentation>
         Root of XML document, holds all message templates
         and their elements
   </xs:documentation></xs:annotation>
   <xs:complexType>
      [See encoding and message elements below.]
      <xs:attribute name="package" type="xs:string" />
      <xs:attribute name="version" type="xs:string"
      use="optional" />
      <xs:attribute name="description" type="xs:string"
      use="optional" />
      <xs:attribute name="byteOrder" default="littleEndian">
         <xs:simpleType>
            <xs:restriction base="xs:token">
               <xs:enumeration value="bigEndian" />
               <xs:enumeration value="littleEndian" />
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
```

```
   </xs:complexType>
</xs:element>
```

Example message schema root element

```
<?xml version="1.0"?>
<sbe:messageSchema
   xmlns:sbe="http://www.fixprotocol.org/ns/simple/1.0"
   package="orders" version="FIX.5.0" description="My message
   dictionary" byteOrder="bigEndian">
```

4.3 Data encodings

4.3.1 Encoding sets

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

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

1. Element <type> defines a simple encoding
2. Element <composite> defines a composite encoding
3. Element <enum> defines an enumeration

4.3.2 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.

<types> element

```
<xs:element name="types" minOccurs="1" maxOccurs="unbounded">
   <xs:annotation><xs:documentation>
         More than one set of types may be provided, e.g.
         built-in and custom.
         Names must be unique across all encoding types.
   </xs:documentation></xs:annotation>
   <xs:complexType>
      <xs:sequence>
         <xs:element name="type" type="sbe:encodedDataType"
               minOccurs="1" maxOccurs="unbounded" />
         <xs:element name="composite"
               type="sbe:compositeDataType"
               minOccurs="0" maxOccurs="unbounded" />
         <xs:element name="enum" type="sbe:enumType"
               minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

4.3.2.1 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"/>
```

4.3.3 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.

4.3.3.1 <type> element content

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

4.3.3.1.1 Constant value

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

4.3.3.2 <type> attributes

<type> attribute	Description	XML type	Usage	Valid values
`name`	Name of encoding	`string`	required	Must be unique among all encoding types in a schema.
`description`	Documentation of the type	`string`	optional	
`presence`	Presence of any field encoded with this type	`token`		`required` `optional` `constant`
`nullValue`	Special value used to indicate null for an optional field	`string`	Only valid if `presence=optional`	The 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	`nonnegative-Integer`	`length` and `variableLength` are mutually exclusive	
`variableLength`	If true, represents a variable length field, e.g. Raw Data	`boolean`		

<type> attribute	Description	XML type	Usage	Valid values

`primitiveType`	The primitive data type that backs the encoding	`token`	required	`char`
				`int8`
				`int16`
				`int32`
				`int64`
				`uint8`
				`uint16`
				`uint32`
				`uint64`

`fixUsage`	Represents a FIX data type	`token`	optional	Same as field `fixUsage` – see below.

4.3.3.3 FIX data type specification

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

<type> element

```
<xs:complexType name="encodedDataType" mixed="true">
      <xs:annotation><xs:documentation>
            Simple data type of a field
         </xs:documentation></xs:annotation>
      <xs:simpleContent>
         <xs:extension base="xs:token">
            <xs:attribute name="name" type="xs:string"
            use="required" />
            <xs:attribute name="description" type="xs:string"
            use="optional" />
            <xs:attribute name="presence" default="required">
               <xs:simpleType>
                  <xs:restriction base="xs:token">
                     <!-- value must always be populated -->
                     <xs:enumeration value="required" />
                     <!-- Value set to nullValue to indicate
                     value not set -->
                     <xs:enumeration value="optional" />
                     <!-- Value does not vary -->
                     <xs:enumeration value="constant" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:attribute>
            <!-- Special values in string format -->
            <!-- Override of default null indicator -->
            <xs:attribute name="nullValue" type="xs:string"
            use="optional" />
            <!-- Valid numerical range -->
            <xs:attribute name="minValue" type="xs:string"
            use="optional" />
            <xs:attribute name="maxValue" type="xs:string"
            use="optional" />
            <!-- Number of elements of the primitiveType -->
```

```
            <xs:attribute name="length"
            type="xs:nonNegativeInteger" default="1" />
            <xs:attribute name="variableLength"
            type="xs:boolean"
               default="false" />
            <xs:attribute name="primitiveType" use="required">
               <xs:simpleType>
                  <xs:restriction base="xs:token">
                     <xs:enumeration value="char" />
                     <xs:enumeration value="int8" />
                     <xs:enumeration value="int16" />
                     <xs:enumeration value="int32" />
                     <xs:enumeration value="int64" />
                     <xs:enumeration value="uint8" />
                     <xs:enumeration value="uint16" />
                     <xs:enumeration value="uint32" />
                     <xs:enumeration value="uint64" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:attribute>
            <xs:attributeGroup ref="sbe:fieldAttributes" />
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
```

Simple type examples

```
<type name="numInGroup" primitiveType="uint8"
fixUsage="NumInGroup" />

<type name="length" primitiveType="uint8" fixUsage="Length"
   maxValue="200" />

<type name="rawData" primitiveType="char" fixUsage="data"
   variableLength="true" />

<type name="GeneralIdentifier" primitiveType="char"
   fixUsage="char" description="Identifies class or source of
   the PartyID" presence="constant">C</type>
```

4.3.4 Composite encodings

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

4.3.4.1 <composite> attributes

<type> attribute	Description	XML type	Usage	Valid values
`name`	Name of encoding	`string`	required	Must be unique among all encoding types.
`description`	Documentation of the type	`string`	optional	

<type> attribute	Description	XML type	Usage	Valid values
`fixUsage`	Represents a FIX data type	`token`	optional	Same as field `fixUsage` – see below.

4.3.4.2 Composite type elements

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

<composite> element

```
<xs:complexType name="compositeDataType" mixed="true">
   <xs:annotation><xs:documentation>
         A derived data type; composed of two or more simple
         types
   </xs:documentation></xs:annotation>
   <xs:sequence>
      <xs:element name="type" type="sbe:encodedDataType"
            minOccurs="2" maxOccurs="unbounded" />
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="description" type="xs:string"
   use="optional" />
   <xs:attributeGroup ref="sbe:fieldAttributes" />
</xs:complexType>
```

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

```
<composite name="decimal32" fixUsage="Price">
   <type name="mantissa" primitiveType="int32" />
   <type name="exponent" primitiveType="int8"
   presence="constant">-4</type>
</composite>
```

4.3.5 Enumeration encodings

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

4.3.5.1 <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	`string`	required	Must be unique among all encoding types.

<enum> attribute	Description	XML type	Usage	Valid values
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.

4.3.5.2 <validValue> element attributes

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

<enum> attribute	Description	XML type	Usage	Valid values
name	Symbolic name of value	string	required	Must be unique among valid values in the enumeration.
description	Documentation of the value	string	optional	

4.3.5.3 < validValue > element content

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

```
<xs:complexType name="enumType" mixed="true">
   <xs:annotation><xs:documentation>
         An enumeration of valid values
   </xs:documentation></xs:annotation>
   <xs:sequence>
      <xs:element name="validValue" type="sbe:validValue"
         minOccurs="1" maxOccurs="unbounded" />
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="encodingType" type="xs:string"
   use="required" />
   <xs:attribute name="description" type="xs:string"
      use="optional" />
</xs:complexType>

<xs:complexType name="validValue">
   <xs:annotation><xs:documentation>
         Valid value as a string
   </xs:documentation></xs:annotation>
   <xs:simpleContent>
      <xs:extension base="xs:token">
         <xs:attribute name="name" type="xs:string"
         use="required" />
         <xs:attribute name="description" type="xs:string"
         use="optional" />
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

Enumeration example (not all valid values listed)
This enumeration is encoded as an 8 bit unsigned integer value. Others are encoded as char codes.

```
<type name="intEnum" primitiveType="uint8" fixUsage="int" />

<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>
```

4.4 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.

4.4.1 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.

4.4.2 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.

4.4.3 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	string	required	
id	Unique 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.

<message> attribute	Description	XML type	Usage	Valid values
fixMsgType	Documents value of FIX MsgType for a message	string	optional	Listed in FIX specifications

```
<xs:element name="message">
   <xs:annotation><xs:documentation>
      A message type
   </xs:documentation></xs:annotation>
   <xs:complexType mixed="true">
      <xs:sequence>
         <xs:annotation><xs:documentation>
            Fixed length fields only here
         </xs:documentation></xs:annotation>
         <xs:element name="field" type="sbe:fieldType"
   minOccurs="0" maxOccurs="unbounded" />
         <xs:element name="group" type="sbe:groupType"
   minOccurs="0" maxOccurs="unbounded" />
         <xs:element name="data" type="sbe:fieldType"
   minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"
         use="required" />
      <xs:attribute name="id" type="xs:unsignedShort"
         use="required" />
      <xs:attribute name="description" type="xs:string"
         use="optional" />
      <!-- Space reserved for root level of message, not
      include groups -->
      <xs:attribute name="blockLength"
      type="xs:nonNegativeInteger" use="optional" />
      <!-- Standard MsgType value, such as 'D' for
      NewOrderSingle message -->
      <xs:attribute name="fixMsgType" type="xs:string"
use="optional" />
   </xs:complexType>
</xs:element>
```

Example <message> element

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

4.5 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	string	required	
id	Unique field	unsigned	required	Must be unique

Schema attribute	Description	XML type	Usage	Valid values
	identifier (FIX tag)	`Short`		within a template
`description`	Documentation	`string`	optional	
`type`	Encoding type name, one of simple type, composite type or enumeration.	`string`	required	Must match the `name` attribute of a simple `<type>`, `<composite>` encoding type or `<enum>`.
`offset`	Offset to the start of the field within a message or repeating group entry. By default, the offset is the sum of preceding field sizes, but it may be increased to effect byte alignment.	`unsigned Int`	optional	Must be greater than or equal to the sum of preceding field sizes.
`presence`	Field presence	`enumera-tion`	Default = `required`	`required` = field value is required; not tested for null. `optional` = field value may be null; null value is default for data type. `constant`= constant value not sent on wire.
`groupName`	Name of a referenced repeating group. Used for NumInGroup fields.	`string`	optional	Must match the `name` attribute of a `<group>` member of the same message.
`refId`	Cross reference to the ID (tag) of another field. Used to tie a Length field to a raw data field, for example.	`unsigned Short`	optional	Must match the `id` attribute of another field in the message.

```
<xs:complexType name="fieldType">
   <xs:annotation><xs:documentation>
      A field of a message of a specified dataType
```

```
   </xs:documentation></xs:annotation>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="id" type="xs:unsignedShort"
   use="required" />
   <!-- A simple or composite encoding type -->
   <xs:attribute name="type" type="xs:string" use="required" />
   <xs:attribute name="refId" type="xs:unsignedShort"
   use="optional" />
   <xs:attribute name="description" type="xs:string"
   use="optional" />
   <xs:attribute name="offset" type="xs:unsignedInt"
   use="optional" />
   <xs:attribute name="groupName" type="xs:string"
   use="optional" />
   <xs:attributeGroup ref="sbe:fieldAttributes" />
   <xs:attributeGroup ref="sbe:subFieldEncodingNames" />
</xs:complexType>
```

Example field schemas

Field that uses a composite encoding

```
<composite name="intQty32" fixUsage="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" />
```

Cross-referenced fields

```
<field type="length" name="EncryptedNewPasswordLen" id="1403"
   refId="1404" />
<data type="rawData" name="EncryptedNewPassword" id="1404" />
```

4.6 Repeating group schema

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

<group> attribute	Description	XML type	Usage	Valid values
name	Name of a group	string	required	
description	Documentation	string	optional	
parentGroup-Name	Documents a logical hierarchy of groups	string	optional	Must match the name attribute of another <group>

<group> attribute	Description	XML type	Usage	Valid values
blockLength	Reserved size in number of octets for a group entry	unsigned Int	optional	if specified, must be greater than or equal to the sum of field lengths.

```
<xs:complexType name="groupType">
   <xs:annotation><xs:documentation>
      A repeating group contains an array of homogeneous
      entries (fixed length)
   </xs:documentation></xs:annotation>
   <xs:sequence>
      <xs:element name="field" type="sbe:fieldType"
      minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="group" type="sbe:groupType"
      minOccurs="0" maxOccurs="unbounded" />
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="description" type="xs:string"
   use="optional" />
   <!-- Documents a logical hierarchy of groups -->
   <xs:attribute name="parentGroupName" type="xs:string"
      use="optional" />
   <!-- Space reserved for an entry -->
   <xs:attribute name="blockLength"
   type="xs:nonNegativeInteger" use="optional" />
</xs:complexType>
```

Example group schema with associated NumInGroup field

```
<field type="uint8" name="NoPartyIDs" id="453"
   groupName="Parties" fixUsage="NumInGroup" />
<group name="Parties">
   <field type="string14" name="PartyID" id="448" />
   <field type="partyRoleEnum" name="PartyRole" id="452" />
</group>
```

4.7 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 encoding	A field or <enum> references a type name that is undefined.
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

Error condition	Error description
Fixed `length` incompatible with `variableLength`	Attributes `length` and `variableLength` are mutually exclusive
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.
Missing `fixUsage`	The FIX data type must be specified on either a field or the encoding that it references.
`fixUsage` 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

4.8 Example message schemas

4.8.1 Message with a repeating group

```
<message name="ListOrder" id="2" description="Simplified
   NewOrderList. Demonstrates repeating group">
   <field name="ListID" id="66" type="string14"
   fixUsage="String"/>
   <field name="BidType" id="394" type="uint8"
   fixUsage="int"/>
   <field name="NoOrders" id="73" type="uint8"
   groupName="ListOrdGrp" fixUsage="NumInGroup"/>
   <group name="ListOrdGrp">
      <field name="ClOrdID" id="11" type="string14"
      fixUsage="String"/>
      <field name="ListSeqNo" id="67" type="uint32"
      fixUsage="int"/>
      <field name="Symbol" id="55" type="string8"
      fixUsage="String"/>
      <field name="Side" id="54" type="char"          `
      fixUsage="char"/>
      <field name="OrderQty" id="38" type="intQty32"
      fixUsage="Qty"/>
   </group>
</message>
```

4.8.2 Message with raw data fields

```
<message name="UserRequest" id="4" description="Demonstrates
   raw data usage">
   <field name="UserRequestId" id="923" type="string14"
   fixUsage="String"/>
```

```
   <field name="UserRequestType" id="924" type="uint8"
   fixUsage="int"/>
   <field name="UserName" id="553" type="string14"
   fixUsage="String"/>
   <field name="Password" id="554" type="string14"
   fixUsage="String"/>
   <field name="NewPassword" id="925" type="string14"
   fixUsage="String"/>
   <field name="EncryptedPasswordMethod" id="1400" type="uint8"
   description="This should be an enum but values undefined."
   fixUsage="int"/>
   <field name="EncryptedPasswordLen" id="1401" type="uint8"
   fixUsage="Length"/>
   <field name="EncryptedNewPasswordLen" id="1403" type="uint8"
   fixUsage="Length"/>
   <field name="RawDataLength" id="95" type="uint8"
   fixUsage="Length"/>
   <data name="EncryptedPassword" id="1402" type="rawData"
   fixUsage="data"/>
   <data name="EncryptedNewPassword" id="1404" type="rawData"
   fixUsage="data"/>
   <data name="RawData" id="96" type="rawData"
   fixUsage="data"/>
</message>
```

5 Usage Guidelines

5.1 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"
   fixUsage="String"/>
```

Example of an identifer field with numeric encoding

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

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