FAST℠ Specification

Version 1.x.1

2006-12-20

Status of this Document

This document specifies a standards protocol for the FIX community, and requests discussion and suggestions for improvements.

Distribution

Distribution of this document is unlimited.

Copyright Notice

Abstract

This document specifies FAST, which is a space and processing efficient encoding method for message oriented data streams. It defines the layout of a binary representation and the semantics of a control structure called a template. It also defines an XML syntax for concrete template definitions.

Disclaimer

No proprietary or ownership interest of any kind is granted with respect to the FIX Protocol (or any rights therein) except as expressly set out in FPL's copyright and acceptable use policy.

Acknowledgements

FPL would like to recognize the efforts of many people in producing this specification. The design is based on ideas from and discussions between Rolf Andersson, Daniel May and Mike Kreutzjans. David Rosenberg authored the specification. Matt Simpson, Richard Shriver, Jim Northey have provided substantial input and feedback to successive drafts of the specification. Anders Furuhed, Yuriy Gormakh, Sitaram Guruswamy, Wei Keok, Göran Forsström, and Mats Ljungqvist have reviewed successive drafts of the specification. Matt Simpson and Greg Orsini contributed the examples section. Finally, the feedback of all Market Data Optimization Work Group participants played an important role in creating this document.

Table of Contents

Document History

Version	Date	Author	Description
1.x.01	2006-05-07	David Rosenborg	Branched into unified specification based on TD extended with parts of TE. Added EBNF grammar to TE section. Added blocked encoding. Renamed groups in TE to segments to minimize conflicts with the other uses of that word. Formalized the concept of stop bit encoded entities.
1.x.02	2006-05-09	David Rosenborg	Initial draft release
1.x.07	2006-08-17	David Rosenberg	FAST Specification 1.1 candidate release and request for comments
1.x.09	2006-12-05	David Rosenborg	Minor clarifications, additions, and inclusion of examples
1.x.1	2006-12-20		Final version approved by the Market Data Optimization Working Group

1 Introduction

This document defines the structure and semantics of FAST, which is a binary encoding method for message oriented data streams. FAST is an acronym for FIX Adapted for Streaming. Although the original purpose of FAST was optimization of FIX messages, the encoding method defined in this document has been generalized to apply to a wider set of protocols.

The encoding method reduces the size of a data stream on two levels. First, a concept referred to as Field Operators allows data affinities of a stream to be leveraged and redundant data to be removed. Second, serialization of the remaining data is accomplished through binary encoding which draws on self-describing field lengths and bit maps indicating the presence or absence of fields.

The encoding is performed with respect to a control structure called a template. A template controls the encoding of a portion of the stream by specifying the order and structure of fields, their field operators and the binary encoding representations to use.

This specification defines a concrete syntax for template definitions. The purpose of the concrete syntax is to provide a normative, full fidelity format that is both human and machine readable. It serves as the default format for authoring, storing and interchanging FAST templates.

The concrete syntax in this specification is however not intended to be used on the wire when two end points exchange template definitions over a FAST session. For wire transfers, the FAST Session Control Protocol [SCP] provides a FAST serialization of the template structures defined by this document.

This document formally defines the concrete syntax as an XML structure using a schema language.

A processor (encoder or decoder) is not required to use the concrete syntax. A processor can for example read template definitions encoded as FAST messages using SCP, or it can even have them hard coded in the program.

A processor typically manages a set of templates. Even though the concrete syntax provides means for defining either a single template or a set of templates in an XML document, this specification does not define how to build or maintain sets or libraries of templates in general. A specific set of templates used by a specific processor can even be defined using multiple simultaneous sources such as XML documents and SCP messages.

The following fragment of XML is an example of a template definition in the concrete syntax.

```
<templates xmlns="http://www.fixprotocol.org/ns/template-definition"
    templateNs="http://www.fixprotocol.org/ns/templates/sample"
    ns="http://www.fixprotocol.org/ns/fix">
 <template name="MDRefreshSample">
  <typeRef name="MarketDataIncrementalRefresh"/>
  <string name="BeginString" id="8"> <constant value="FIX4.4"/> </string>
  <string name="MessageType" id="35"> <constant value="X"/> </string>
  <string name="SenderCompID" id="49"> <copy/> </string>
  <uInt32 name="MsgSeqNum" id="34"> <increment/> </uInt32>
  <sequence name="MDEntries">
   <length name="NoMDEntries" id="268"/>
   <uInt32 name="MDUpdateAction" id="279"> <copy/> </uInt32>
   <string name="MDEntryType" id="269"> <copy/> </string>
   <string name="Symbol" id="55"> <copy/> </string>
   <string name="SecurityType" id="167"> <copy/> </string>
   <decimal name="MDEntryPx" id="270"> <delta/> </decimal>
   <decimal name="MDEntrySize" id="271"> <delta/> </decimal>
   <uInt32 name="NumberOfOrders" id="346"> <delta/> </uInt32>
   <string name="QuoteCondition" id="276"> <copy/> </string>
   <string name="TradeCondition" id="277"> <copy/> </string>
  </sequence>
 </template>
</templates>
```

Appendix 3 contains more examples of the concrete syntax together with examples of the encoding of the corresponding fields.

2 Terminology

The term *encode* refers to the process of serializing an instance of an application type to a FAST stream.

The term *decode* refers to the process of deserializing a part of a FAST stream into an instance of an application type. This document does not always explicitly describe a decoding operation when it follows trivially from the defined encoding operation.

This document defines encoding and decoding of FAST streams in terms of what may be considered a processing model. This model is however to be treated as abstract. Implementations are free to implement FAST encoding and decoding in any way as long as the result is the same as if this model was used.

3 Notation

This document uses the compact syntax of the RELAX NG schema language [RNC] to formally define the XML structure for template definitions. Fragments of the schema are interleaved with descriptive text. The complete schema that is available in Appendix 1 is extensible. In the schema fragments provided in the text, the parts related to extensibility have been left out. A W3C XML Schema [XSD] version is provided in Appendix 2.

Errors are labeled with an error identifier in brackets.

References are labeled with the corresponding identifier from the References section in brackets.

3.1 XML Namespace

The Template Definition (TD) namespace has the URI "http://www.fixprotocol.org/ns/fast/td/1.1". The prefix `td:` is used throughout this document for referring to elements in this namespace.

```
default namespace = "http://www.fixprotocol.org/ns/fast/td/1.1"
```

4 Error Handling

An error that is detected by solely examining a template definition is referred to as a *static error*. Encoders and decoders must signal static errors and the template where the error occurred must be discarded.

When reading template definitions from an XML document it is a static error [ERR S1] if the document

- is not well-formed as defined in Extensible Markup Language [XML],
- does not conform to the constraints in Namespaces in XML [XMLNS],
- is not valid according to the schema as specified in Appendix 1.

An error that is detected when encoding or decoding a FAST stream is referred to as either a *dynamic* or *reportable error*. Encoders and decoders must signal dynamic errors and are encouraged to signal reportable errors but may refrain from doing so. A typical reason for not signaling reportable errors can be to achieve better performance. To ensure interoperability, it is however recommended that all errors are signaled during development and testing of implementations.

5 Application Types

An *application type* represents a type of a group or message in the application using FAST. This document does not specify the structure of application types and treat them as abstract entities, but assumes that they can be mapped to the following model.

- A *group* is a named type comprising an unordered set of fields.

- A *field* has a name and a type. The name must be unique within the group. The type can be a primitive type, a sequence type or a group type.
- A *sequence* comprises a length and an ordered set of elements. Each element is of group type. This specification does not require that all elements have identical group types. This may result in heterogeneous sequences at the application level. A particular application of FAST may however constrain this and require that all elements have the same type.
- The *primitive* types are ASCII string, Unicode string, uInt32, int32, uInt64, int64, decimal and byte vector. The value domains of these are the same as for the corresponding types defined in this document.
- A group appearing at the topmost level of a stream is also referred to as a *message*.

6 Templates

A *template* specifies how to encode an instance of an application type, or part thereof, as a stream of bytes.

Each template is identified by a name that is used when referring to a template either from the definition of another template, or in an external context.

Templates do not constitute types in themselves but are associated with application types by reference. There can be more than one template per application type[1]. It is also possible to create a template that can be used for more than one application type.

A template is defined by the `<td:template>` element in the concrete syntax. A template definition XML document can either contain a single template or a collection of templates. A collection of templates must be enclosed in a `<td:templates>` element. This element can hold namespace attributes applicable to the whole enclosed set of templates.

A template contains a sequence of *instructions*. The order of the instructions is significant and corresponds to the order of the data in the stream. There are two categories of instructions: *field instructions* and *template reference instructions*. Field instructions specify how to encode fields of the instance to the stream. Template reference instructions provide means for defining parts of a template by reference to other templates.

```
start = templates | template
templates = element templates { nsAttr?, templateNsAttr?, dictionaryAttr?, template* }
template = element template { templateNsName, nsAttr?, dictionaryAttr?, typeRef?, instruction* }
instruction = field | templateRef
```

6.1 Instruction Context

Encoding and decoding takes place in the context of an instruction. The context consists of:

- a set of templates
- a *current template*
- a set of application types
- a *current application type*
- a set of dictionaries
- an optional initial value

The current application type is initially the special type *any*. The current application type changes when the processor encounters an element containing a `<td:typeRef>` element. The new type is applicable to the

[1] Having multiple templates for the same application type makes it possible to optimize for different uses of the type. As an example, many FIX messages contain a wide variety of fields where only a few are used in combination. Using different templates for each major combination is a more compact alternative to having one template with many optional members.

instructions contained within the element. The <td:typeRef> can appear in the <td:template>; <td:group> and <td:sequence> elements.

```
typeRef = element typeRef { nameAttr, nsAttr?}
```

The current template is a reference to the template being processed. It is updated when a template identifier is encountered in the stream. A static template reference can also change the current template as described in the Template Reference Instruction section.

The dictionary set and initial value are described in the Operators section below.

6.2 Field Instructions

Each field instruction has a name and a type. The name identifies the corresponding field in the current application type. The type specifies the basic encoding of the field. It is a dynamic error [ERR D1] if the type of a instruction cannot be converted to, or from when decoding, the type of the corresponding application field. See the section Type Conversion for permitted conversions.

The optional presence attribute indicates whether the field is mandatory or optional. If the attribute is not specified, the field is mandatory.

A primitive field, i.e. a field that is not a group or sequence, can have a field operator. The operator specifies an optimization operation for the field.

```
field = integerField | decimalField | asciiStringField | unicodeStringField | byteVectorField | sequence | group
fieldInstrContent = nsName, presenceAttr?, fieldOp?
presenceAttr = attribute presence { "mandatory" | "optional" }
```

6.2.1 Integer Field Instructions

Integer Numbers have unlimited size in the transfer encoding. However, applications typically use fixed sizes for integers. An *integer* field instruction must therefore specify the bounds of the integer. The number in the element name indicates the size in bits of the integer field instruction. The encoding and decoding of a value is not affected by the size of the integer.

A leading "int" indicates that the field is signed and "uInt" indicates that the field is unsigned.

```
integerField =
  element int32 { fieldInstrContent }
  | element uInt32 { fieldInstrContent }
  | element int64 { fieldInstrContent }
  | element uInt64 { fieldInstrContent }
```

It is a dynamic error [ERR D2] if an integer in the stream is greater than the maximum value or less than the minimum value for the specified type. The following table lists the minimum and maximum values that can be represented by each integer type:

Type	Min	Max
int32	-2147483648	2147483647
uInt32	0	4294967295
int64	-9223372036854775808	9223372036854775807
uInt64	0	18446744073709551615

6.2.2 Decimal Field Instructions

A *decimal* field instruction indicates that the field is represented by two parts: an exponent and a mantissa. The instruction can contain a field operator for the whole decimal or individual operators for the two parts. If an operator is specified individually for the mantissa and/or the exponent, the operators are applied individually to each part before the decimal number is combined. The field operator operands for

the exponent and mantissa are signed integers, *int32* and *int64* respectively. If no operator is specified or if a single operator is specified for the entire decimal, the operand is a decimal number and is represented as a Scaled Number in the transfer encoding.

Even though the exponent is treated as an *int32*, its allowed value range is [-63 … 63]. It is a reportable error [ERR R1] if the exponent falls outside of this range after any operator has been applied.

When operators are applied individually, the exponent and mantissa parts have generated names unique to the name of the decimal field. These names will be used as default values for the keys of the corresponding operators.

If the decimal field has optional presence and has individual operators, the presence of the mantissa is dependent on the presence of the exponent. See the section Presence Map and NULL Utilization for the definition.

When using individual operators it is possible to limit the range and precision of a decimal. If for example the constant operator is used for the exponent with a constant value of 2, it is not possible to encode 0.01 in this field. It is a dynamic error [ERR D3] if the value cannot be encoded in the field due to limitations introduced by the use of an operator.

An initial value specified on an operator for a decimal field instruction will be normalized. This is described in the Initial Values section below.

```
decimalField = element decimal { nsName, presenceAttr?, ( fieldOp | decFieldOp ) }
decFieldOp = element exponent { fieldOp }?, element mantissa { fieldOp }?
```

6.2.3 String Field Instruction

A *string* field instruction has an optional `charset` attribute indicating the character set used in the string. There are two supported character sets: ASCII and Unicode, indicated by the attribute values “ascii” and “unicode” respectively. If the attribute is not specified, the character set is ASCII. Depending on the specified character set, the string is represented as an ASCII String or Unicode String in the transfer encoding. If the character set is Unicode, an optional <td:length> element can be specified to associate a name with the length preamble of the underlying byte vector.

```
asciiStringField = element string { fieldInstrContent, attribute charset { "ascii" }? }
unicodeStringField = element string { byteVectorLength?, fieldInstrContent, attribute charset { "unicode" } }
```

6.2.4 Byte Vector Field Instruction

A *byte vector* field instruction indicates that the field is represented as a Byte Vector in the transfer encoding.

In the concrete syntax it is possible to associate a name with the length preamble of a byte vector by specifying a `<td:length>` element. Logically this field is of type uInt32. The use of `<td:length>` does not change how a byte vector is encoded in the stream, it just serves as a handle for the processor to report the length back to an application.

```
byteVectorField = element byteVector { byteVectorLength?, fieldInstrContent }
byteVectorLength = element length { nsName }
```

6.2.5 Sequence Field Instruction

A *sequence* field instruction specifies that the field in the application type is of sequence type and that the contained group of instructions should be used repeatedly to encode each element. If any instruction of the group needs to allocate a bit in a presence map, each element is represented as a *segment* in the transfer encoding.

A sequence has an associated length field containing an unsigned integer indicating the number of encoded elements. When a length field is present in the stream, it must appear directly before the encoded

elements. The length field has a name, is of type *uInt32* and can have a field operator. There are two styles of naming:

- *implicit* – the name is generated and is unique to the name of the sequence field. The name is guaranteed to never collide with a field name explicitly specified in a template.
- *explicit* – the name is explicitly specified in the template definition.

A sequence can be mandatory or optional. An optional sequence means that the length field is optional.

A sequence instruction is represented by the `<td:sequence>` element in the concrete syntax. It can have an optional `<td:length>` child element, preceding any instructions. This element specifies the properties of the length field. If it has a name attribute, the naming is explicit, otherwise it is implicit.

If no `<td:length>` element is specified, the length field has an implicit name and no field operator.

```
sequence = element sequence { nsName, presenceAttr?, dictionaryAttr?, typeRef?, length?, instruction* }
length = element length { nsName?, fieldOp? }
```

6.2.6 Group Field Instruction

A *group* field instruction associates a name and presence attribute with a group of instructions. If any instruction of the group needs to allocate a bit in a presence map, the group is represented as a *segment* in the transfer encoding.

It is not required that the current application type has a corresponding notion of a group. This means that the fields resulting from decoding the group can possibly be flattened into one layer in the application type.

The main purpose of the group field instruction is to enable a single bit in the presence map to indicate the presence of a whole group of fields.

```
group = element group { nsName, presenceAttr?, dictionaryAttr?, typeRef?, instruction* }
```

6.3 Field Operators

Field operators specify ways to optimize the encoding of a field. Not all operators are applicable to all field types. These constraints are however not always expressed in the schema, but are expressed in the descriptive text for each operator. It is a static error [ERR S2] if an operator is specified for a field type for which it is not applicable.

```
fieldOp = constant | \default | copy | increment | delta | tail
```

6.3.1 Dictionaries and Previous Values

Some operators rely on a *previous value*. Previous values are maintained in named *dictionaries*. A dictionary has a set of entries. Each entry has a name and a typed value. The value can be in one of three states: *undefined*, *empty* and *assigned*. All values are in the state undefined when processing starts. The state assigned indicates that the previous value is present and empty indicates that it is absent. The state empty is only applicable to optional fields. See the Presence Map and NULL Utilization section for details on when empty is set.

The previous value in a dictionary for an operator is the value of the entry with the same name as its *key*. The default key of an operator is the name of its field. An explicit key can be specified by the `key` attribute. By specifying explicit keys, operators for fields with different names can share the same previous value entry in a dictionary.

It is a dynamic error [ERR D4] if the field of an operator accessing an entry does not have the same type as the value of the entry.

The dictionary name is specified by the `dictionary` attribute on the field operator element or where allowed by the schema on ancestor elements. If there are more than one `dictionary` attribute in the ancestry, the attribute of the nearest element applies. If the attribute is not specified, the global dictionary is used.

There are three predefined dictionaries:

- *template* – the dictionary is local to the current template. This means that an operator in template *T1* will share the same dictionary as an operator in template *T2* iff[2] *T1* = *T2*.
- *type* – the dictionary is local to the current application type. This means that an operator in template *T1* that is a template of application type *A1* will share the same dictionary as an operator in template *T2* that is a template of application type *A2* iff *A1* = *A2*.
- *global* – the dictionary is global. All operators share the same dictionary regardless of the template and application type.

All other dictionaries are referred to as user defined. Two operators will share the same user defined dictionary iff they specify identical dictionary names.

A dictionary can be explicitly *reset*. Resetting a dictionary will set the state of all its entries to undefined. This specification does not define how a reset is signaled to a decoder or encoder. It is however important that resets appear in the same order in the encoder and decoder with respect to the order of the content of the stream.

```
opContext = dictionaryAttr?, nsKey?, initialValueAttr?
dictionaryAttr = attribute dictionary { "template" | "type" | "global" | string }
nsKey = keyAttr, nsAttr?
keyAttr = attribute key { token }
```

6.3.2 Initial Values

An initial value is specified by the `value` attribute on the operator element. The value is a string of Unicode characters. This value is converted to the type of the field as defined in the Converting from String section below. The possible dynamic and reportable errors that may occur during conversion are treated as static errors [ERR S3] when interpreting the initial value.

If the field is of type decimal, the value resulting from the conversion is normalized. The reason for this is that the exponent and mantissa must be predictable when operators are applied to them individually. A decimal value is normalized by adjusting the mantissa and exponent so that the integer remainder after dividing the mantissa by 10 is not zero: `mant % 10 ≠ 0`. For example `100 * 10^0` would be normalized as `1 * 10^2`. If the mantissa is zero, the normalized decimal has a zero mantissa and a zero exponent.

```
initialValueAttr = attribute value { text }
```

6.3.3 Constant Operator

The *constant* operator specifies that the value of a field will always be the same. The value of the field is the initial value. It is a static error [ERR S4] if the instruction context has no initial value.

The value of a constant field is never transferred.

The constant operator is applicable to all field types.

```
constant = element constant { initialValueAttr }
```

[2] if and only if

6.3.4 Default Operator

The *default* operator specifies that the value of a field is either present in the stream or it will be the initial value. Unless the field has optional presence, it is a static error [ERR S5] if the instruction context has no initial value. If the field has optional presence and no initial value, the field is considered absent when there is no value in the stream.

The default operator is applicable to all field types.

```
\default = element default { initialValueAttr? }
```

6.3.5 Copy Operator

The *copy* operator specifies that the value of a field is optionally present in the stream. If the value is present in the stream it becomes the new previous value.

When the value is not present in the stream there are three cases depending on the state of the previous value:

- assigned – the value of the field is the previous value.
- undefined – the value of the field is the initial value that also becomes the new previous value. Unless the field has optional presence, it is a dynamic error [ERR D5] if the instruction context has no initial value. If the field has optional presence and no initial value, the field is considered absent and the state of the previous value is changed to empty.
- empty – the value of the field is empty. If the field is optional the value is considered absent. It is a dynamic error [ERR D6] if the field is mandatory.

The copy operator is applicable to all field types.

```
copy = element copy { opContext }
```

6.3.6 Increment Operator

The *increment* operator specifies that the value of a field is optionally present in the stream. If the value is present in the stream it becomes the new previous value.

When the value is not present in the stream there are three cases depending on the state of the previous value:

- assigned – the value of the field is the previous value incremented by one. The incremented value also becomes the new previous value.
- undefined – the value of the field is the initial value that also becomes the new previous value. Unless the field has optional presence, it is a dynamic error [ERR D5] if the instruction context has no initial value. If the field has optional presence and no initial value, the field is considered absent and the state of the previous value is changed to empty.
- empty – the value of the field is empty. If the field is optional, the value is considered absent. It is a dynamic error [ERR D6] if the field is mandatory.

The increment operator is applicable to *integer* field types.

An integer is incremented by adding one to it. If the value is the maximum value of the type it becomes the minimum value after the increment.

```
increment = element increment { opContext }
```

6.3.7 Delta Operator

The *delta* operator specifies that a *delta value* is present in the stream. If the field has optional presence, the delta value can be NULL. In that case the value of the field is considered absent. Otherwise the field is obtained by *combining* the delta value with a *base value*.

```
delta = element delta { opContext }
```

The base value depends on the state of the previous value in the following way:

- assigned – the base value is the previous value.
- undefined – the base value is the initial value if present in the instruction context. Otherwise a type dependant *default base value* is used.
- empty – it is a dynamic error [ERR D6] if the previous value is empty.

The following sections define the delta value representations, the default base values and how values are combined depending on type.

6.3.7.1 Delta for Integers

The delta value is represented as an Integer Delta in the transfer encoding. The combined value is the sum of the base and delta values.

The default base value for integers is 0.

It is a reportable error [ERR R4] if the combined value is less than the minimum value or greater than the maximum value of the specific integer type.

> **NOTE**: The size of the integer required for the delta may be larger than the specified size for the field type. For example, if a field of type *uInt32* has the base 4294967295 and the new value is 17 an *int64* is required to represent the delta -4294967278. However, this does not affect how the delta appears in the stream.

6.3.7.2 Delta for Decimal

The delta value is represented as a Scaled Number Delta in the transfer encoding. The combined value is calculated by individually adding the exponent and the mantissa of the delta to their base value counterparts.

It is a reportable error [ERR R1] if the combined exponent is less than -63 or greater than 63, or if the combined mantissa exceeds the value range of an *int64*.

The default base value for decimal is 0. The exponent of the default base value is 0.

> **NOTE**: Since the delta operator for decimals comprises individual deltas for the exponent and mantissa, an implementation must store the previous value of a decimal in such a way that the layout in exponent and mantissa parts is preserved or can be recreated when processing the next field.

6.3.7.3 Delta for ASCII Strings

The delta value is represented as an ASCII String Delta in the transfer encoding. The subtraction length of the delta specifies the number of characters to remove from the front or back of the base value. Characters are removed from the front when the subtraction length is negative. The string part of the delta value represents the characters to add to the same end of the base value as specified by the sign of the subtraction length.

The subtraction length uses an excess-1 encoding: if the value is negative when decoding, it is incremented by one to get the number of characters to subtract. This makes it possible to encode negative zero as -1, which can be used to encode an operation that adds to the front without removing any characters.

The default base value is the empty string

It is a dynamic error [ERR D7] if the subtraction length is larger than the number of characters in the base value, or if it does not fall in the value range of an *int32*.

6.3.7.4 Delta for Unicode Strings

The delta value for Unicode strings is structurally equivalent with the delta value for byte vectors with the additional constraint that the content of the byte vector in the delta is UTF-8 bytes. The delta operates on the encoded bytes as opposed to the Unicode characters. As a consequence, a delta value may end in an incomplete UTF-8 byte sequence. It is a reportable error [ERR R2] if the combined value is not a valid UTF-8 sequence.

6.3.7.5 Delta for Byte Vectors

The delta is represented as a Byte Vector Delta in the transfer encoding. The subtraction length of the delta specifies the number of bytes to remove from the front or back of the base value. Bytes are removed from the front when the subtraction length is negative. The byte vector part of the delta value represents the bytes to add to the same end of the base value as specified by the sign of the leading subtraction length.

The subtraction length uses the same excess-1 encoding as for ASCII strings: if the value is negative when decoding, it is incremented by one to get the number of characters to subtract.

The default base value is the empty byte vector.

It is a dynamic error [ERR D7] if the subtraction length is larger than the number of bytes in the base value, or if it exceeds the value range of an *int32*.

6.3.8 Tail Operator

The *tail* operator specifies that a *tail value* is optionally present in the stream.

If the field has optional presence, the tail value can be NULL. In that case the value of the field is considered absent. Otherwise, if the tail value is present, the value of the field is obtained by *combining* the tail value with a *base value*.

The base value depends on the state of the previous value in the following way:

- assigned – the base value is the previous value.
- undefined – the base value is the initial value if present in the instruction context. Otherwise a type dependant *default base value* is used.
- empty – the base value is the initial value if present in the instruction context. Otherwise a type dependant *default base value* is used.

The combined value becomes the new previous value.

If the tail value is not present in the stream, the value of the field depends on the state of the previous value in the following way:

- assigned – the value of the field is the previous value.
- undefined – the value of the field is the initial value that also becomes the new previous value. Unless the field has optional presence, it is a dynamic error [ERR D6] if the instruction context has no initial value. If the field has optional presence and no initial value, the field is considered absent and the state of the previous value is changed to empty.
- empty – the value of the field is empty. If the field is optional the value is considered absent. It is a dynamic error [ERR D7] if the field is mandatory.

In the concrete syntax the tail operator is represented by the `<td:tail>` element:

```
tail = element tail { opContext }
```

The following sections define the tail value representations, the default base values and how values are combined depending on type. The tail operator is only applicable to these types.

6.3.8.1 Tail for ASCII Strings

The tail value is represented as an ASCII String in the transfer encoding. The length of the string specifies the number of characters to remove from the back of the base value. The tail value represents the characters to append to the remaining string

If the length of the tail value exceeds the length of the base value, the combined value becomes the tail value.

The default base value is the empty string

6.3.8.2 Tail for Unicode Strings

The tail value for Unicode strings is structurally equivalent with the tail value for byte vectors with the additional constraint that the content of the byte vector in the delta is UTF-8 bytes. The tail operator operates on the encoded bytes as opposed to the Unicode characters. As a consequence, a tail value may end in an incomplete UTF-8 byte sequence. It is a reportable error [ERR R2] if the combined value is not a valid UTF-8 sequence.

6.3.8.3 Tail for Byte Vectors

The tail value is represented as a Byte Vector in the transfer encoding. The length of the tail value specifies the number of bytes to remove from the back of the base value. The tail value represents the bytes to append to the remaining byte vector.

If the length of the tail value exceeds the length of the base value, the combined value becomes the tail value.

The default base value is the empty byte vector.

6.4 Template Reference Instruction

The template reference instruction specifies that a part of the template is specified by another template. A template reference can be either static or dynamic. A reference is static when a name is specified in the instruction. Otherwise it is dynamic.

A static reference specifies that processing should continue with the referred template as the current template. A static reference does not imply that there is a presence map or template identifier in the stream. It is a dynamic error [ERR D8] if no template exists with the specified name.

A dynamic reference specifies that a presence map and a template identifier are present in the stream. The processing continues with the template indicated by the identifier as the current template. The representation in the transfer encoding is a *segment*. It is a dynamic error [ERR D9] if no template is associated with the template identifier appearing in the stream.

When processing reaches the end of the referred static or dynamic template, it continues at the point after the referring instruction and the current template is restored.

```
templateRef = element templateRef { ( nameAttr, templateNsAttr? )? }
```

7 Names

A name in a template definition consists of two parts, a namespace URI and a local name.

The namespace URI for application types, fields and operator keys is specified by the `ns` attribute which can appear either on the same element as the local name or on any ancestor element. If there are more than one `ns` attribute in the ancestry, the attribute of the nearest element applies. If no `ns` attribute is specified the namespace URI is the empty string.

The namespace URI for templates is specified by the `templateNs` attribute that is inherited in the same way as the `ns` attribute. The reason for having a separate attribute for template names is that message and field names often share the same standardized namespace whereas template names are likely to be put in a vendor specific namespace.

The fact that a namespace is a URI does not mean that it must point to a resource. The motivation of a URI in this context is simply to constrain the syntax and to encourage the use of, for example, company or organization URLs to make namespaces universally unique.

The attribute `name` specifies a local name.

Two names are equal iff their namespace identifiers are equal and their local names are equal.

```
nsName = nameAttr, nsAttr?, idAttr?
templateNsName = nameAttr, templateNsAttr?, idAttr?
nameAttr = attribute name { token }
nsAttr = attribute ns { text }
templateNsAttr = attribute templateNs { text }
idAttr = attribute id { token }
```

7.1 Auxiliary Identifiers

Any component that can have a name and is not a reference can also have an auxiliary identifier. The identifier is specified by the `id` attribute. This specification does not define any semantics for the use of auxiliary identifiers. Nor does it specify the scope of identifiers. However, a particular communication protocol adapted to FAST may choose to constrain the use of auxiliary identifiers.

For example, when using FIX over FAST, a typical use of auxiliary identifiers would be to specify the FIX tag number on each field. Another possible use of auxiliary identifiers would be to assign static template identifiers to be used in the communication between two parties that do not support dynamic template exchange and identifier assignment.

> **NOTE**: The fact that this specification provides a way to specify auxiliary identifiers in-band does not imply that there cannot be other in-band (through foreign elements or attributes, see the Extensibility section) or out-of-band schemes for mapping auxiliary identifiers to names of components. The `id` attribute is provided for convenience.

8 Type Conversion

When the type of a field in a template differs from the type of the corresponding field in the current application type, values must be converted when the field is encoded and decoded. This section defines the conversion between pairs of different types.

Byte vectors can only be converted to and from strings. All other conversion with byte vectors are dynamic errors [ERR D10].

8.1 Converting from String

In the following sections the term *whitespace trimmed* refers to the operation where any leading or trailing whitespace is removed before the string is interpreted. The following characters in ASCII hexadecimal are considered to be whitespace: 20 (space), 09 (horizontal tab), 0D (carriage return), and 0A (linefeed).

It is a dynamic error [ERR D11] if a string does not match the syntax specified by the following sections.

NOTE: Although the syntax for signed integers and decimals allows negative zeroes like -0 and -0.0, these values will be normalized to their positive counterparts. Negative zeroes cannot be represented in a FAST stream.

8.1.1 Converting to Integers

The string is interpreted as a sequence of digits '0' – '9'. If the type is signed, a leading minus is allowed to indicate a negative number. The literal is whitespace trimmed. It is a reportable error [ERR R4] if the resulting number does not fit within the specified size of the integer. The string "4711" would for example cause an error if the type was *int8*.

8.1.2 Converting to Decimal

The string has an integer part and a decimal part. It is allowed to specify either or both of them. If both are specified, a decimal period must appear between them. If only the decimal part is specified, a decimal period must appear before it. A leading minus sign indicates a negative number. Example: 1, 1.1, .1 and -0.1 are allowed representations. The literal is whitespace trimmed. It is a reportable error [ERR R1] if the conversion would result in an exponent less than -63 or greater than 63 or if the mantissa does not fit in the range of an *int64*.

8.1.3 Converting to Byte Vector

The string is interpreted as an even number of hexadecimal digits [0-9A-Fa-f] possibly interleaved with whitespace. The literal is turned into a byte vector by first stripping any whitespace. Then each pair of characters is interpreted as a hexadecimal number representing a single byte.

8.1.4 Converting Between Character Sets

An ASCII string is trivially converted to a Unicode string since ASCII is a subset of Unicode. A Unicode string can be converted to an ASCII string if it only contains ASCII characters. Otherwise it is a reportable error [ERR R3].

8.2 Converting from Integers

8.2.1 Converting to Integers

Integers of different types can be converted to each other as long as there is no loss of precision. It is a reportable error [ERR R4] if the value cannot be represented in the target type. A negative value can for example not be converted to an unsigned type.

8.2.2 Converting to Decimal

An integer can be converted to decimal if it can be represented as a scaled number with an exponent in the range [-63 … -63] and an *int64* mantissa. Otherwise it is a reportable error [ERR R1].

8.2.3 Converting to String

The number is represented as a sequence of digits '0' – '9'. The number must not have any leading zeroes. If the type is signed and the number is negative the sequence of digits is preceded by a minus sign ('-').

8.3 Converting from Decimal

8.3.1 Converting to Integers

A decimal can be converted to an integer iff it has no decimal part. That is, the value is in fact an integer. It is a reportable error [ERR R5] if the value is in fact not an integer.

8.3.2 Converting to String

If the number is in fact an integer, it is converted as if was of integer type. Otherwise the number is represented by an integer part and a decimal part separated by a decimal point ('.'). Each part is a sequence of digits '0' – '9'. There must be at least one digit on each side of the decimal point. If the number is negative it is preceded by a minus sign ('-'). The integer part must not have any leading zeroes.

8.4 Converting from Byte Vector to String

The byte vector is represented as a sequence of an even number of hexadecimal digits [0-9a-f]. Each pair is a hexadecimal number representing a byte in the vector.

9 Extensibility

Application specific data can be added to a template definition in the XML format by using foreign attributes and elements. Any element in the schema may have foreign attributes and foreign child elements. A foreign attribute is an attribute with a name whose namespace URI is neither the empty string nor the TD namespace URI. A foreign element is an element with a name whose namespace URI is not the TD namespace URI. Foreign child elements may be placed freely with respect to other child elements. There are no restrictions on the content of foreign attributes and elements.

The extensibility parts are implemented in the schema using the *other* pattern which is placed at the relevant places in the schema using the interleave operator.

```
other = foreignAttr*, foreignElm*
foreignElm = element * - td:* { any }
foreignAttr = attribute * - (local:* | td:*) { text }
any = attribute * { text }*, ( text | element * { any } )*
```

10 Transfer Encoding

The following EBNF grammar specifies the overall structure of a FAST stream. Terminal symbols are in italics. The start symbol is stream.

```
stream      ::=  message* | block*
block       ::=  BlockSize message+
message     ::=  segment
segment     ::=  PresenceMap TemplateIdentifier? (field | segment)*
field       ::=  integer | string | delta | ScaledNumber | ByteVector
integer     ::=  UnsignedInteger | SignedInteger
string      ::=  ASCIIString | UnicodeString
delta       ::=  IntegerDelta | ScaledNumberDelta | ASCIIStringDelta |
                 ByteVectorDelta
```

A FAST *stream* consists of a sequence of messages or a sequence of blocks. This specification does not provide a way of saying which style is used for a particular stream. Thus this must be agreed upon between the producer and consumer of the encoded data.

A *block* is a sequence of one or more messages. A block has a leading block size specifying the number of bytes occupied by the messages of the block. It is a dynamic error [ERR D12] if a block has zero size. The

block size is represented as an Unsigned Integer which may be overlong. The overlong property is defined in the Integer Numbers section below.

Each *message* is represented as a segment, a *message segment*.

A *segment* has a header consisting of a Presence Map followed by an optional Template Identifier. The segment has a template identifier either if it is a message segment, or if the segment appears as the result of a dynamic template reference instruction. A template identifier is encoded as if a copy operator was specified. The operator uses the global dictionary and has an internal key common to all template identifier fields. This means that a segment with a template identifier does not always contain the template identifier physically. However, the first bit in the presence map is allocated by its copy operator.

The body of a segment is a sequence of *fields* and possible sub segments. The extent of a segment is defined by the template and is dependent on the settings in the presence map.

10.1 Byte and Bit Ordering

All integer fields are represented using the big-endian convention, where bits and bytes are in network byte order, where high order bits precede low order bits, and high order bytes precede low order bytes.

10.2 Stop Bit Encoded Entities

An important property of the FAST transfer encoding is the use of *stop bit encoded entities*. A stop bit encoded entity is a sequence of bytes where the most significant bit in each byte indicates whether the next byte is part of the entity. If the bit is not set, the next byte belongs to the entity, otherwise it is the last byte. The seven bits following the stop bit are *significant data bits*. The *entity value* of a stop bit encoded entity is the concatenation of the significant data bits of each byte. The number of bits in the entity value is always a multiple of seven. The minimum length of an entity value is seven bits.

10.3 Template Identifier

A template identifier is represented as an Unsigned Integer in the stream. It is a reportable error [ERR R6] if it is overlong.

It is a dynamic error [ERR D9] if a decoder cannot find a template associated with a template identifier appearing in the stream.

This specification does not define how to map an identifier to the name of a template. A particular implementation may choose to use statically allocated template identifiers. In this case, auxiliary identifiers in the concrete syntax can be used to convey the mapping. Other implementations may choose to allocate template identifiers dynamically using for example the Session Control Protocol [SCP].

10.4 Nullability

Each field has a type that has a *nullability* property. If a type is *nullable*, there is a special representation of a NULL value. When a type is *non-nullable*, no representation for NULL is reserved. All nullable types are constructed in such a way that NULL is represented as a 7-bit entity value where all bits are zero. It is represented as 0x80 when stop bit encoded.

Unless explicitly specified, non-nullable representations are used.

10.5 Presence Map

A presence map is a sequence of bits. Fields of the segment of the presence map utilize the bits as specified by the current template.

A presence map is represented as a stop bit encoded entity. Logically a presence map has an infinite suffix of zeroes. This makes it possible to truncate a presence map that ends in a sequence where the bits are all zero. The length of the remaining part must be a multiple of seven.

A presence map is overlong if it has more than seven bits and ends in seven or more bits that are all zero. It is a reportable error [ERR R7] if a presence map is overlong. It is a reportable error [ERR R8] if a presence map contains more bits than required by the instructions that utilize it.

10.5.1 Presence Map and NULL Utilization

Bits in the presence map are allocated in field entry order. That is, instructions appearing earlier in a template will allocate bits of higher order than those appearing later. Bits are allocated in the presence map of the current segment.

> **NOTE**: The type of a length field of a sequence is *uInt32*, as specified by section 6.2.5. This means that any encoding rule that is applicable to an unsigned integer field is also applicable to the length field of a sequence.

A field will not occupy any bit in the presence map if it is mandatory and has the constant operator.

An optional field with the constant operator will occupy a single bit. If the bit is set, the value is the initial value in the instruction context. If the bit is not set, the value is considered absent.

If a field is mandatory and has no field operator, it will not occupy any bit in the presence map and its value must always appear in the stream.

If a group field is optional, it will occupy a single bit in the presence map. The contents of the group may appear in the stream iff the bit is set. The instructions in the group are not processed if the bit is not set. This means that the previous values of fields in the group are not affected by an absent group. If the application representation of a message has no notion of groups, each field of an absent group is considered absent.

If a field is optional and has no field operator, it is encoded with a nullable representation and the NULL is used to represent absence of a value. It will not occupy any bits in the presence map.

The default, copy, and increment operators have the following presence map and NULL utilization:

- Mandatory integer, decimal, string and byte vector fields – one bit. If set, the value appears in the stream.
- Optional integer, decimal, string and byte vector fields – one bit. If set, the value appears in the stream in a nullable representation. A NULL indicates that the value is absent and the state of the previous value is set to empty, except when the default operator is used in which case the state of the previous value is left unchanged.

The delta operator has the following presence map utilization:

- Mandatory integer, decimal, string and byte vector fields – no bit.
- Optional integer, decimal, string and byte vector fields – no bit. The delta appears in the stream in a nullable representation. A NULL indicates that the delta is absent. Note that the previous value is *not* set to empty but is left untouched if the value is absent.

The tail operator has the following presence map utilization:

- Mandatory string and byte vector fields – one bit.
- Optional string and byte vector fields – one bit. The tail value appears in the stream in a nullable representation. A NULL indicates that the value is absent and the state of the previous value is set to empty.

Decimal fields with individual operators have the following utilization:

- If the decimal has mandatory presence, the exponent and mantissa fields are treated as two separate mandatory integer fields as described above.
- If the decimal has optional presence, the exponent field is treated as an optional integer field and the mantissa field is treated as a mandatory integer field. The presence of the mantissa field and

any related bits in the presence map are dependent on the presence of the exponent. The mantissa field appears in the stream iff the exponent value is considered present. If the mantissa has an operator that requires a bit in the presence map, this bit is present iff the exponent value is considered present.

10.6 Fields

10.6.1 Integer Numbers

Integers are represented as stop bit encoded entities. An integer is *overlong* if the entity value still represents the same integer after removing seven or more of the most significant bits. Unless the integer is used as a block size, it is a reportable error [ERR R6] if an overlong integer number appears in the stream.

If an integer is nullable, every non-negative integer is incremented by 1 before it is encoded. The NULL representation of a nullable integer is a 7-bit entity value where all bits are zero.

NOTE: The encoding of a nullable integer may require more bits than the maximum number of bits specified in the integer type. For example, the nullable representation of the maximum 32 bit unsigned integer 4294967295 is 4294967296, which would be encoded as 0x10 0x00 0x00 0x00 0x80, and requires 33 significant bits in the entity value.

10.6.1.1Signed Integer

The entity value is a two's complement integer representation [TWOC]. The most significant data bit of the entity value is the sign bit.

NOTE: Since the most significant bit of the entity value is the sign bit, there are some situations where the seven most significant bits of the entity value must all be zeroes. For example, if the value to encode is 64, which has the binary representation 01000000, the stop bit encoding must be 0x00 0xC0. If we did not have the extra seven leading zero bits, the most significant bit, which is also the sign bit, would be one, and thus the encoding would incorrectly represent -64.

10.6.1.2Unsigned Integer

The entity value is the binary representation of the integer.

10.6.2 Scaled Number

Scaled numbers, like floating point numbers are represented as a mantissa and an exponent.

Floating point numbers use a base-2 exponent for computational efficiency reasons, while scaled numbers use a base-10 exponent in order to support exact representation of decimal numbers.

$$number = mant * 10^{exp}$$

The numerical value is obtained by multiplying the mantissa with the base-10 power of the exponent.

A scaled number is represented as a Signed Integer exponent followed by a Signed Integer mantissa.

If a scaled number is nullable, the exponent is nullable and the mantissa is non-nullable. A NULL scaled number is represented as a NULL exponent. The mantissa is present in the stream iff the exponent is not NULL.

10.6.3 ASCII String

An ASCII String is represented as a stop bit encoded entity. The entity value is interpreted as a sequence of 7-bit ASCII characters.

A sequence of bits starting with seven zero bits is referred to as having a *zero-preamble*. A string that starts with a zero-preamble consists of the bits that remain after removing the preamble. A string is overlong if

there are bits left after removing the preamble and the first seven of those bits are not all zero. It is a reportable error [ERR R9] if an overlong string appears in the stream.

If a string has a zero-preamble and there are no bits left after removing the preamble, it represents the empty string.

If an ASCII String is nullable, an additional zero-preamble is allowed at the start of the string. The bits that follow are interpreted as a non-nullable string, including a possible zero preamble. If there are no remaining bits after removing the preamble the value represents the NULL string.

The following table summarizes the use of zero-preambles:

Entity value	Nullable	Description
0x00		Empty string
0x00 0x00		"\0"
0x00 0x41		"A", Overlong
0x00	Yes	NULL
0x00 0x00	Yes	Empty String
0x00 0x41	Yes	"A", Overlong
0x00 0x00 0x00	Yes	"\0"
0x00 0x00 0x41	Yes	"A", Overlong

10.6.4 Unicode String

A Unicode String is represented as a Byte Vector containing the UTF-8 encoded representation of the string. UTF-8 is defined in the Unicode 3.2 [UNICODE] standard. If a Unicode String is nullable, it is represented by a nullable Byte Vector.

10.6.5 Byte Vector

A byte vector field is represented as an Unsigned Integer size preamble followed by the specified number of raw bytes. Each byte in the data part has eight significant data bits. As a consequence, the data part is not stop bit encoded.

A nullable byte vector has a nullable size preamble. The NULL byte vector is represented by a NULL size preamble.

10.7 Delta

10.7.1 Integer Delta

The delta value is represented as a Signed Integer. A nullable integer delta is represented by a nullable Signed Integer.

10.7.2 Scaled Number Delta

The delta value is represented as two Signed Integers. The first integer is the delta for the exponent and the second is the delta for the mantissa. If the delta is nullable, it has a nullable exponent delta and a non-nullable mantissa delta. A NULL delta is represented as a NULL exponent delta. The mantissa delta is present in the stream iff the exponent delta is not NULL.

10.7.3 ASCII String Delta

The delta value is represented as a Signed Integer subtraction length followed by an ASCII String. If the delta is nullable, the subtraction length is nullable. A NULL delta is represented as a NULL subtraction length. The string part is present in the stream iff the subtraction length is not NULL.

10.7.4 Byte Vector Delta

The delta value is represented as a Signed Integer subtraction length followed by a Byte Vector. If the delta is nullable, the subtraction length is nullable. A NULL delta is represented as a NULL subtraction length. The byte vector part is present in the stream iff the subtraction length is not NULL.

Appendix 1 RELAX NG Schema

```
default namespace td = "http://www.fixprotocol.org/ns/fast/td/1.1"
namespace local = ""

start = templates | template

templates = element templates { ( nsAttr?, templateNsAttr?, dictionaryAttr?, template* ) & other }

template = element template { ( templateNsName, nsAttr?, dictionaryAttr?, typeRef?, instruction* ) & other }

typeRef = element typeRef { nameAttr, nsAttr?, other }

instruction = field | templateRef

fieldInstrContent = ( nsName, presenceAttr?, fieldOp? ) & other

field = integerField | decimalField | asciiStringField | unicodeStringField | byteVectorField | sequence | group

integerField =
element int32 { fieldInstrContent }
| element uInt32 { fieldInstrContent }
| element int64 { fieldInstrContent }
| element uInt64 { fieldInstrContent }

decimalField = element decimal { ( nsName, presenceAttr?, ( fieldOp | decFieldOp ) ) & other }

decFieldOp = element exponent { fieldOp & other }?, element mantissa { fieldOp & other }?

asciiStringField = element string { fieldInstrContent, attribute charset { "ascii" }? }
unicodeStringField = element string { byteVectorLength?, fieldInstrContent, attribute charset { "unicode" } }

byteVectorField = element byteVector { byteVectorLength?, fieldInstrContent }
byteVectorLength = element length { nsName }

sequence = element sequence { ( nsName, presenceAttr?, dictionaryAttr?, typeRef?, length?, instruction* ) & other }

length = element length { ( nsName?, fieldOp? ) & other }

group = element group { ( nsName, presenceAttr?, dictionaryAttr?, typeRef?, instruction* ) & other }

fieldOp = constant | \default | copy | increment | delta | tail

constant = element constant { initialValueAttr & other }

\default = element default { initialValueAttr? & other }

copy = element copy { opContext }

increment = element increment { opContext }

delta = element delta { opContext }

tail = element tail { opContext }

initialValueAttr = attribute value { text }

opContext = ( dictionaryAttr?, nsKey?, initialValueAttr? ) & other

dictionaryAttr = attribute dictionary { "template" | "type" | "global" | string }

nsKey = keyAttr, nsAttr?

keyAttr = attribute key { token }

templateRef = element templateRef { ( nameAttr, templateNsAttr? )?, other }

presenceAttr = attribute presence { "mandatory" | "optional" }

nsName = nameAttr, nsAttr?, idAttr?

templateNsName = nameAttr, templateNsAttr?, idAttr?

nameAttr = attribute name { token }

nsAttr = attribute ns { text }

templateNsAttr = attribute templateNs { text }

idAttr = attribute id { token }

other = foreignAttr*, foreignElm*

foreignElm = element * - td:* { any }

foreignAttr = attribute * - (local:* | td:*) { text }

any = attribute * { text }*, ( text | element * { any } )*
```

Appendix 2 W3C XML Schema (Non-Normative)

This schema is an automatic translation of the normative RELAX NG schema and is only an approximation of the original. This is because XML Schema is less expressive than RELAX NG. As a result, this schema is more permissive than the original.

```
<xs:schema xmlns:td="http://www.fixprotocol.org/ns/fast/td/1.1" xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://www.fixprotocol.org/ns/fast/td/1.1">

 <xs:element name="templates">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="td:template"/>
    <xs:group ref="td:other"/>
   </xs:choice>
   <xs:attribute name="ns"/>
   <xs:attribute name="templateNs"/>
   <xs:attribute name="dictionary">
    <xs:simpleType>
     <xs:union memberTypes="xs:string">
      <xs:simpleType>
       <xs:restriction base="xs:token">
        <xs:enumeration value="template"/>
        <xs:enumeration value="type"/>
        <xs:enumeration value="global"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:union>
    </xs:simpleType>
   </xs:attribute>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="template">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:choice>
     <xs:element ref="td:typeRef"/>
     <xs:group ref="td:instruction"/>
    </xs:choice>
    <xs:group ref="td:other"/>
   </xs:choice>
   <xs:attributeGroup ref="td:templateNsName"/>
   <xs:attribute name="ns"/>
   <xs:attribute name="dictionary">
    <xs:simpleType>
     <xs:union memberTypes="xs:string">
      <xs:simpleType>
       <xs:restriction base="xs:token">
        <xs:enumeration value="template"/>
        <xs:enumeration value="type"/>
        <xs:enumeration value="global"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:union>
    </xs:simpleType>
   </xs:attribute>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="typeRef">
  <xs:complexType>
   <xs:group ref="td:other"/>
   <xs:attributeGroup ref="td:nameAttr"/>
   <xs:attribute name="ns"/>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
 </xs:element>

 <xs:group name="instruction">
  <xs:choice>
   <xs:group ref="td:field"/>
   <xs:element ref="td:templateRef"/>
  </xs:choice>
 </xs:group>

 <xs:group name="fieldInstrContent">
  <xs:sequence>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="td:fieldOp"/>
    <xs:group ref="td:other"/>
   </xs:choice>
  </xs:sequence>
 </xs:group>

 <xs:attributeGroup name="fieldInstrContent">
  <xs:attributeGroup ref="td:nsName"/>
  <xs:attribute name="presence">
   <xs:simpleType>
    <xs:restriction base="xs:token">
```

```
      <xs:enumeration value="mandatory"/>
      <xs:enumeration value="optional"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
  <xs:attributeGroup ref="td:other"/>
</xs:attributeGroup>

<xs:group name="field">
  <xs:choice>
   <xs:group ref="td:integerField"/>
   <xs:element ref="td:decimal"/>
   <xs:group ref="td:stringField"/>
   <xs:element ref="td:byteVector"/>
   <xs:element ref="td:sequence"/>
   <xs:element ref="td:group"/>
  </xs:choice>
</xs:group>

<xs:complexType name="integerField">
  <xs:group ref="td:fieldInstrContent"/>
  <xs:attributeGroup ref="td:fieldInstrContent"/>
</xs:complexType>

<xs:group name="integerField">
  <xs:choice>
   <xs:element ref="td:int32"/>
   <xs:element ref="td:uInt32"/>
   <xs:element ref="td:int64"/>
   <xs:element ref="td:uInt64"/>
  </xs:choice>
</xs:group>

<xs:element name="int32" type="td:integerField"/>
<xs:element name="uInt32" type="td:integerField"/>
<xs:element name="int64" type="td:integerField"/>
<xs:element name="uInt64" type="td:integerField"/>

<xs:element name="decimal">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:choice>
     <xs:group ref="td:fieldOp"/>
     <xs:choice>
      <xs:element ref="td:exponent"/>
      <xs:element ref="td:mantissa"/>
     </xs:choice>
    </xs:choice>
    <xs:group ref="td:other"/>
   </xs:choice>
   <xs:attributeGroup ref="td:nsName"/>
   <xs:attribute name="presence">
    <xs:simpleType>
     <xs:restriction base="xs:token">
      <xs:enumeration value="mandatory"/>
      <xs:enumeration value="optional"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
</xs:element>

<xs:element name="exponent">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="td:fieldOp"/>
    <xs:group ref="td:other"/>
   </xs:choice>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
</xs:element>

<xs:element name="mantissa">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="td:fieldOp"/>
    <xs:group ref="td:other"/>
   </xs:choice>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
</xs:element>

<xs:group name="stringField">
  <xs:sequence>
   <xs:element name="string">
    <xs:complexType>
     <xs:sequence>
      <xs:group minOccurs="0" ref="td:byteVectorLength"/>
      <xs:group ref="td:fieldInstrContent"/>
     </xs:sequence>
     <xs:attributeGroup ref="td:fieldInstrContent"/>
     <xs:attribute name="charset">
      <xs:simpleType>
       <xs:restriction base="xs:token">
        <xs:enumeration value="ascii"/>
```

```
          <xs:enumeration value="unicode"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:attribute>
     </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>

<xs:element name="byteVector">
  <xs:complexType>
    <xs:sequence>
      <xs:group minOccurs="0" ref="td:byteVectorLength"/>
      <xs:group ref="td:fieldInstrContent"/>
    </xs:sequence>
    <xs:attributeGroup ref="td:fieldInstrContent"/>
  </xs:complexType>
</xs:element>

<xs:group name="byteVectorLength">
  <xs:sequence>
    <xs:element name="length">
      <xs:complexType> <xs:attributeGroup ref="td:nsName"/> </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>

<xs:element name="sequence">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:choice>
        <xs:element ref="td:typeRef"/>
        <xs:group ref="td:length"/>
        <xs:group ref="td:instruction"/>
      </xs:choice>
      <xs:group ref="td:other"/>
    </xs:choice>
    <xs:attributeGroup ref="td:nsName"/>
    <xs:attribute name="presence">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="mandatory"/>
          <xs:enumeration value="optional"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="dictionary">
      <xs:simpleType>
        <xs:union memberTypes="xs:string">
          <xs:simpleType>
            <xs:restriction base="xs:token">
              <xs:enumeration value="template"/>
              <xs:enumeration value="type"/>
              <xs:enumeration value="global"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:union>
      </xs:simpleType>
    </xs:attribute>
    <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
</xs:element>

<xs:group name="length">
  <xs:sequence>
    <xs:element name="length">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:group ref="td:fieldOp"/>
          <xs:group ref="td:other"/>
        </xs:choice>
        <xs:attribute name="name" type="xs:token"/>
        <xs:attribute name="ns"/>
        <xs:attribute name="id" type="xs:token"/>
        <xs:attributeGroup ref="td:other"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>

<xs:element name="group">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:choice>
        <xs:element ref="td:typeRef"/>
        <xs:group ref="td:instruction"/>
      </xs:choice>
      <xs:group ref="td:other"/>
    </xs:choice>
    <xs:attributeGroup ref="td:nsName"/>
    <xs:attribute name="presence">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="mandatory"/>
          <xs:enumeration value="optional"/>
        </xs:restriction>
```

```
    </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="dictionary">
    <xs:simpleType>
     <xs:union memberTypes="xs:string">
      <xs:simpleType>
       <xs:restriction base="xs:token">
        <xs:enumeration value="template"/>
        <xs:enumeration value="type"/>
        <xs:enumeration value="global"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:union>
    </xs:simpleType>
   </xs:attribute>
   <xs:attributeGroup ref="td:other"/>
  </xs:complexType>
</xs:element>

<xs:group name="fieldOp">
 <xs:choice>
  <xs:element ref="td:constant"/>
  <xs:element ref="td:default"/>
  <xs:element ref="td:copy"/>
  <xs:element ref="td:increment"/>
  <xs:element ref="td:delta"/>
  <xs:element ref="td:tail"/>
 </xs:choice>
</xs:group>

<xs:element name="constant">
 <xs:complexType>
  <xs:group ref="td:other"/>
  <xs:attributeGroup ref="td:initialValueAttr"/>
  <xs:attributeGroup ref="td:other"/>
 </xs:complexType>
</xs:element>

<xs:element name="default">
 <xs:complexType>
  <xs:group ref="td:other"/>
  <xs:attribute name="value"/>
  <xs:attributeGroup ref="td:other"/>
 </xs:complexType>
</xs:element>

<xs:element name="copy" type="td:opContext"/>
<xs:element name="increment" type="td:opContext"/>
<xs:element name="delta" type="td:opContext"/>
<xs:element name="tail" type="td:opContext"/>

<xs:attributeGroup name="initialValueAttr"> <xs:attribute use="required" name="value"/> </xs:attributeGroup>

<xs:complexType name="opContext">
 <xs:group ref="td:other"/>
 <xs:attribute name="dictionary">
  <xs:simpleType>
   <xs:union memberTypes="xs:string">
    <xs:simpleType>
     <xs:restriction base="xs:token">
      <xs:enumeration value="template"/>
      <xs:enumeration value="type"/>
      <xs:enumeration value="global"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:union>
  </xs:simpleType>
 </xs:attribute>
 <xs:attribute name="key" type="xs:token"/>
 <xs:attribute name="ns"/>
 <xs:attribute name="value"/>
 <xs:attributeGroup ref="td:other"/>
</xs:complexType>

<xs:element name="templateRef">
 <xs:complexType>
  <xs:group ref="td:other"/>
  <xs:attribute name="name" type="xs:token"/>
  <xs:attribute name="templateNs"/>
  <xs:attributeGroup ref="td:other"/>
 </xs:complexType>
</xs:element>

<xs:attributeGroup name="nsName">
 <xs:attributeGroup ref="td:nameAttr"/>
 <xs:attribute name="ns"/>
 <xs:attribute name="id" type="xs:token"/>
</xs:attributeGroup>

<xs:attributeGroup name="templateNsName">
 <xs:attributeGroup ref="td:nameAttr"/>
 <xs:attribute name="templateNs"/>
 <xs:attribute name="id" type="xs:token"/>
</xs:attributeGroup>

<xs:attributeGroup name="nameAttr"> <xs:attribute use="required" name="name" type="xs:token"/> </xs:attributeGroup>
```

```
 <xs:group name="other">
  <xs:sequence> <xs:group minOccurs="0" ref="td:foreignElm" maxOccurs="unbounded"/> </xs:sequence>
 </xs:group>

 <xs:attributeGroup name="other"> <xs:attributeGroup ref="td:foreignAttr"/> </xs:attributeGroup>

 <xs:group name="foreignElm">
  <xs:choice>
   <xs:any namespace="##other" processContents="skip"/>
   <xs:any namespace="##local" processContents="skip"/>
  </xs:choice>
 </xs:group>

 <xs:attributeGroup name="foreignAttr"> <xs:anyAttribute namespace="##other" processContents="skip"/> </xs:attributeGroup>
</xs:schema>
```

Appendix 3 Examples (Non-Normative)

Appendix 3.1 Data Type Examples

Appendix 3.1.1 FAST v1.1 Signed Integer Examples

1. Int32 Example – Optional Positive Number

```
<int32 id="1" presence="optional" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
942755	0x0e 0x62 0xa3 00001110 01100010 10100011	0x39 0x45 0xa4 **0**0111001 **0**1000101 **1**0100100
		Stop bits indicated in bold Sign bits indicated by underline Increment by 1 since field is optional and value is non-negative

2. Int32 Example – Mandatory Positive Number

```
<int32 id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
942755	0x0e 0x62 0xa3 00001110 01100010 10100011	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
		Stop bits indicated in bold Sign bits indicated by underline Do not increment by 1 since field is mandatory

3. Int32 Example – Optional Negative Number

```
<int32 id="1" presence="optional" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
-942755	0xf1 0x9d 0x5d 11110001 10011101 01011101	0x46 0x3a 0xdd **0**1000110 **0**0111010 **1**1011101
	High-values are dropped on left-most byte	Stop bits indicated in bold Sign bits indicated by underline Do not increment by 1 since value is negative

4. Int32 Example – Mandatory Negative Number

```
<int32 id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
-7942755	0xff 0x86 0xcd 0x9d 11111111 10000110 11001101 10011101	0x7c 0x1b 0x1b 0x9d **0**1111100 **0**0011011 **0**0011011 **1**0011101

	High-values are dropped on left-most bit	Stop bits indicated in bold Sign bits indicated by underline Do not increment by 1 since field is mandatory

5. Int32 Example – Mandatory Positive Number with sign-bit extension

```
<int32 id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
8193	0x20 0x01 00100000 00000001	0x00 0x40 0x81 ***0***0***00000*** **0**1000000 **1**0000001
		Stop bits indicated in bold Sign bits indicated by underline Sign bit extension necessary to specify sign (italics) Do not increment by 1 since field is mandatory

6. Int32 Example – Mandatory Negative Number with sign-bit extension

```
<int32 id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
-8193	0xff 0xdf 0xff 11111111 11011111 11111111	0x73 0x3f 0xff ***0***1***111111*** **0**0111111 **1**1111111
		Stop bits indicated in bold Sign bits indicated by underline Sign bit extension necessary to specify sign (italics) Do not increment by 1 since field is mandatory

Appendix 3.1.2 FAST v1.1 Unsigned Integer Examples

1. uInt32 Example – Optional Number

```
<uInt32 id="1" presence="optional" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
null	n/a	0x80 **1**0000000
0	0x00 0	0x81 **1**0000001
1	0x01 1	0x82 **1**0000010
942755	0x0e 0x62 0xa3 1110 01100010 10100011	0x39 0x45 0xa4 **0**0111001 **0**1000101 **1**0100100
		Increment by 1 since field is optional Stop bits indicated in bold

2. uInt32 Example – Mandatory Number

```
<uInt32 id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
0	0x00 0	0x80 **1**0000000
1	0x01 1	0x81 **1**0000001
942755	0x0e 0x62 0xa3 1110 01100010 10100011	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
		Do not increment by 1 since field is mandatory Stop bits indicated in bold

Appendix 3.1.3 FAST v1.1 String Examples

1. US ASCII string Example – Optional String

```
<string id="1" presence="optional" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
Null	n/a	0x80 **1**0000000
ABC	0x41 0x42 0x43 01000001 01000010 01000011	0x41 0x42 0xc3 **0**1000001 **0**1000010 **1**1000011
"" – zero length string	n/a	0x00 0x80 **0**0000000 **1**0000000
		Stop bits indicated in bold

2. US ASCII string Example – Mandatory String

```
<string id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary
ABC	0x41 0x42 0x43 01000001 01000010 01000011	0x41 0x42 0xc3 **0**1000001 **0**1000010 **1**1000011
"" – zero length string	n/a	0x80 **1**0000000
		Stop bits indicated in bold

Appendix 3.1.4 FAST v1.1 Byte Vector Examples

1. byteVector Example – Optional byteVector

```
<byteVector id="1" presence="optional" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary	
		Length	Value
Null	n/a	0x80 **1**0000000	n/a
ABC	0x41 0x42 0x43 01000001 01000010 01000011	0x84 **1**0000011	0x41 0x42 0x43 01000001 01000010 01000011
zero length value	n/a	0x81 **1**0000001	n/a
		Increment zero length by 1 since field is optional Stop bits indicated in bold	

2. byteVector Example – Mandatory byteVector

```
<byteVector id="1" presence="mandatory" name="Value"/>
```

Input Value	Native Hex/Binary	FAST Hex/Binary	
		Length	Value
ABC	0x41 0x42 0x43 01000001 01000010 01000011	0x83 **1**0000011	0x41 0x42 0x43 01000001 01000010 01000011
zero length value	n/a	0x80 **1**0000000	n/a
			Stop bits indicated in bold

Appendix 3.1.5 FAST v1.1 Decimal Examples

1. Decimal Example – Mandatory Positive Decimal

```
<decimal id="1" presence="mandatory" name="Value"/>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	Exponent	Mantissa	Exponent	Mantissa
94275500	2	942755	0x82 **1**0000010	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
			Stop bits indicated in bold Sign bits indicated by underline	

2. Decimal Example – Mandatory Positive Decimal with Scaled Mantissa

```
<decimal id="1" presence="mandatory" name="Value"/>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	Exponent	Mantissa	Exponent	Mantissa
94275500	1	9427550	0x81 **1**0000001	0x04 0x3f 0x34 0xde **0**0000100 **0**0111111 **0**0110100 **1**1011110
			Stop bits indicated in bold Sign bits indicated by underline Exponent is set to 1 for scaling mantissa value	

3. Decimal Example – Optional Positive Decimal

```
<decimal id="1" presence="optional" name="Value"/>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	Exponent	Mantissa	Exponent	Mantissa
94275500	2	942755	0x83 **1**0000011	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
			Stop bits indicated in bold Sign bits indicated by underline Increment Exponent by 1 since field is optional and	

		value is non-negative

4. Decimal Example – Mandatory Positive Decimal

```
<decimal id="1" presence="mandatory" name="Value"/>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	**Exponent**	**Mantissa**	**Exponent**	**Mantissa**
9427.55	-2	942755	0xfe **1**1111110	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
			Stop bits indicated in bold Sign bits indicated by underline	

5. Decimal Example – Optional Negative Decimal

```
<decimal id="1" presence="optional" name="Value"/>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	**Exponent**	**Mantissa**	**Exponent**	**Mantissa**
-9427.55	-2	-942755	0xfe **1**1111110	0x46 0x3a 0xdd **0**1000110 **0**0111010 **1**1011101
			Stop bits indicated in bold Sign bits indicated by underline	

6. Decimal Example – Optional Positive Decimal with single field operator

```
<decimal id="1" presence="optional" name="Value"> <copy/> </decimal>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	**Exponent**	**Mantissa**	**Exponent**	**Mantissa**
9427.55	-2	942755	0xfe **1**1111110	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
			Stop bits indicated in bold Sign bits indicated by underline Single Pmap slot is used since field is optional and operator is specified at field level	

7. Decimal Example – Optional Positive Decimal with individual field operators

```
<decimal id="1" presence="optional" name="Value">
 <exponent> <copy/> </exponent>
 <mantissa> <delta/> </mantissa>
</decimal>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	**Exponent**	**Mantissa**	**Exponent**	**Mantissa**
9427.55	-2	942755	0xfe **1**1111110	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
			Stop bits indicated in bold Sign bits indicated by underline Exponent uses a Pmap Mantissa does not use a Pmap slot	

8. Decimal Example – Optional Negative Decimal with sign bit extension

```
<decimal id="1" presence="optional" name="Value"/>
```

Input Value	Decomposed Input Value		FAST Hex/Binary	
Ascii	**Exponent**	**Mantissa**	**Exponent**	**Mantissa**
-8.193	-3	-8193	0xfd **1**1111101	0x73 0x3f 0xff ***01111111*** **0**0111111 **1**1111111
			Stop bits indicated in bold Sign bits indicated by underline Sign bit extension necessary to specify sign (italics)	

Appendix 3.2 Field Operator Examples

Appendix 3.2.1 FAST v1.1 Constant Operator Examples

1. Constant Operator Example – Mandatory Unsigned Integer

```
<uInt32 id="1" presence="mandatory" name="Flag"> <constant value="0"/> </uInt32>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
0	N/A	None	Not Required	None
99	N/A	Error	Error	Error
None	N/A	Error	Error	Error

2. Constant Operator Example – Optional Unsigned Integer

```
<uInt32 id="1" presence="optional" name="Flag"> <constant value="0"/> </uInt32>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
0	N/A	None	1*	None
None	N/A	None	0*	None

* An optional field with the constant operator will occupy a single bit. The bit will be set on if the input value is equal to the initial value specified in the instruction context. The bit will be set off if the input value is absent.

Appendix 3.2.2 FAST v1.1 Default Operator Examples

1. Default Operator Example – Mandatory Unsigned Integer

```
<uInt32 id="1" presence="mandatory" name="Flag"> <default value="0"/> </uInt32>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
0	N/A	None	0	None
1	N/A	1	1	0x81 **1**0000001

2. Default Operator Example for NULL– Optional Unsigned Integer

```
<uInt32 id="1" presence="optional" name="Flag"> <default/> </uInt32>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
None	N/A	None	0	None

Appendix 3.2.3 FAST v1.1 Copy Operator Examples

1. Copy Operator Example – Mandatory String

```
<string id="1" presence="mandatory" name="Flag"> <copy/> </string>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
CME	None	CME	1	0x43 0x4d 0xc5 **0**1000011 **0**1001101 **1**1000101
CME	CME	None	0	None
ISE	CME	ISE	1	0x49 0x53 0xc5 **0**1001001 **0**1010011 **1**1000101

2. Copy Operator Example for NULL – Optional String

```
<string id="1" presence="optional" name="Flag"> <copy/> </string>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
None	None	Null	1	0x80 **1**0000000
None	Null	None	0	None
CME	Null	CME	1	0x43 0x4d 0xc5 **0**1000011 **0**1001101 **1**1000101

Appendix 3.2.4 FAST v1.1 Increment Operator Examples

1. Increment Operator Example – Mandatory Unsigned Integer

```
<uInt32 id="1" presence="mandatory" name="Flag"> <increment value="1"/> </uInt32>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
1	1	None	0	None
2	1	None	0	None
4	2	4	1	0x84 **1**0000100
5	4	None		

Appendix 3.2.5 FAST v1.1 Delta Operator Examples

1. Delta Operator Example – Mandatory Signed Integer

```
<int32 id="1" presence="mandatory" name="Price"> <delta/> </int32>
```

Input Value	Prior Value	Encoded Value	Pmap Bit	FAST Hex/Binary
942755	0	942755	N/A	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
942750	942755	-5	N/A	0xfb **1**1111011
942745	942750	-5	N/A	0xfb **1**1111011
942745	942745	0	N/A	0x80 **1**0000000

The initial prior value in example 1 above is 0 (zero). The default value can be specified in the template.

2. Delta Operator Example – Mandatory Decimal

```
<decimal id="1" presence="mandatory" name="Price"> <delta/> </decimal>
```

Input Value	Prior Value		Encoded Value		Pmap Bit	FAST Hex/Binary	
	Exp	Mant	Exp	Mant		Exponent	Mantissa
9427.55	0	0	-2	942755	N/A	0xfe **1**1111110	0x39 0x45 0xa3 **0**0111001 **0**1000101 **1**0100011
9427.51	-2	942755	0	-4	N/A	0x80 **1**0000000	0xfc **1**1111100
9427.46	-2	942751	0	-5	N/A	0x80 **1**0000000	0xfb **1**1111011

3. Delta Operator Example – Mandatory Decimal with Initial Value

```
<decimal id="1" presence="mandatory" name="Price"> <delta value="12000"/> </decimal>
```

Initial/ Input Value	Prior Value (mantissa)		Encoded Value		Pmap Bit	FAST Hex/Binary	
	Exp	Mant	Exp	Mant		Exp	Mantissa
12000	0	0	N/A	N/A	N/A	N/A	N/A
12100	3	12	-2	1198	N/A	0xfe **1**1111110	0x09 0xae **0**0001001 **1**0101110
12150	1	1210	0	5	N/A	0x80 **1**0000000	0x85 **1**0000101
12200	1	1215	0	5	N/A	0x80 **1**0000000	0x85 **1**0000101

4. Delta Operator Example – Mandatory String

```
<string id="1" presence="mandatory" name="Security"> <delta/> </string>
```

Input Value	Prior Value	Encoded Value	Subtraction Length	Pmap Bit	FAST Hex/Binary	
					Length	Encoded String
GEH6	Empty	GEH6	0	N/A	0x80	0x47 0x45 0x48 0xb6

	string				**1**0000000	**0**1000111 **0**1000101 **0**1001000 **1**0110110
GEM6	GEH6	M6	2	N/A	0x82 **1**0000010	0x4d 0xb6 **0**1001101 **1**0110110
ESM6	GEM6	ES	-2	N/A	0xfd **1**1111101	0x45 0xd3 **0**1000101 **1**1010011
RSESM6	ESM6	RS	-0	N/A	0xff **1**1111111	0x52 0xd3 **0**1010010 **1**1010011

A negative subtraction length is used to remove values from the front of the string. Negative zero is used to append values to the front of the string

The initial prior value is the empty string if no default string has been specified in the template.

Appendix 3.2.6 FAST v1.1 Extended Example

3. Multiple Pmap Slot Example – Optional Positive Decimal with individual field operators

```
<decimal id="1" presence="optional" name="Value">
 <exponent> <copy/> </exponent>
 <mantissa> <copy/> </mantissa>
</decimal>
```

Input Value	Decomposed Input Value				Pmap Bits	FAST Hex/Binary	
	Exponent		Mantissa			Exponent	Mantissa
Ascii	Value	Encoded	Value	Encoded			
9427.55	-2 (no previous)	-2	942755 (no previous)	942755	11	0xfe **1**1111110	0x39 0x45 0xa4 **0**0111001 **0**1000101 **1**0100100
9427.60	-2	None	942760	942760	01	None	0x39 0x45 0xa8 **0**0111001 **0**1000101 **1**0101000
None	NULL	NULL	None	None	1	0x80 **1**0000000	None
					Stop bits indicated in bold Sign bits indicated by underline Exponent and Mantissa each use 1 Pmap slot due to use of separate field operators		

Appendix 3.3 Presence Map

Appendix 3.3.1 FAST v1.1 Presence Map Examples

The following table summarizes the presence bit utilization rules for the different field operators.

	Presence Map Bit is Required	
Operator	**Mandatory**	**Optional**
None	No	No
<constant/>	No	Yes*
<copy/>	Yes	Yes
<default/>	Yes	Yes
<delta/>	No	No
<increment/>	Yes	Yes

<tail/>	Yes	Yes

* An optional field with the constant operator will occupy a single bit. The bit will be set on if the input value is equal to the initial value specified in the instruction context. The bit will be set off if the input value is absent.

Appendix 4 Summary of Error Codes

Static Errors

ERR S1

It is a static error if templates encoded in the concrete XML syntax are in fact not well-formed, do not follow the rules of XML namespaces or are invalid with respect to the schema in Appendix 1.

ERR S2

It is a static error if an operator is specified for a field of a type to which the operator is not applicable.

ERR S3

It is a static error if an initial value specified by the value attribute in the concrete syntax cannot be converted to a value of the type of the field.

ERR S4

It is a static error if no initial value is specified for a constant operator.

ERR S5

It is a static error if no initial value is specified for a default operator on a mandatory field.

Dynamic Errors

ERR D1

It is a dynamic error if type of a field in a template cannot be converted to or from the type of the corresponding application field.

ERR D2

It is a dynamic error if an integer in the stream does not fall within the bounds of the specific integer type specified on the corresponding field.

ERR D3

It is a dynamic error if a decimal value cannot be encoded due to limitations introduced by using individual operators on exponent and mantissa.

ERR D4

It is a dynamic error if the type of the previous value is not the same as the type of the field of the current operator.

ERR D5

It is a dynamic error if a mandatory field is not present in the stream, has an undefined previous value and there is no initial value in the instruction context.

ERR D6

It is a dynamic error if a mandatory field is not present in the stream and has an empty previous value.

ERR D7

It is a dynamic error if the subtraction length exceeds the length of the base value or if it does not fall in the value rang of an *int32*.

ERR D8

It is a dynamic error if the name specified on a static template reference does not point to a template known by the encoder or decoder.

ERR D9

It is a dynamic error if a decoder cannot find a template associated with a template identifier appearing in the stream.

ERR D10

It is a dynamic error to convert byte vectors to and from other types than strings.

ERR D11

It is a dynamic error if the syntax of a string does not follow the rules for the type converted to.

ERR D12

It is a dynamic error if a block length preamble is zero.

Dynamic Errors

ERR R1

It is a reportable error if a decimal cannot be represented by an exponent in the range [-63 … 63] or if the mantissa does not fit in an *int64*.

ERR R2

It is a reportable error if the combined value after applying a tail or delta operator to a Unicode string is not a valid UTF-8 sequence.

ERR R3

It is a reportable error if a Unicode string that is being converted to an ASCII string contains characters that are outside the ASCII character set.

ERR R4

It is a reportable error if a value of an integer type cannot be represented in the target integer type in a conversion.

ERR R5

It is a reportable error if a decimal being converted to an integer has a negative exponent or if the resulting integer does not fit the target integer type.

ERR R6

It is a reportable error if an integer appears in an overlong encoding.

ERR R7

It is a reportable error if a presence map is overlong.

ERR R8

It is a reportable error if a presence map contains more bits than required.

ERR R9

It is a reportable error if a string appears in an overlong encoding.

References

SCP

Rolf Andersson. *FAST Session Control Protocol Specification*. FPL, 2005.

RNC

James Clark, editor. *RELAX NG Compact Syntax*. OASIS, 2002.

XSD

Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, editors. *XML Schema Part 1*. W3C (World Wide Web Consortium), 2001.

XML

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, editors. *Extensible Markup Language*. W3C (World Wide Web Consortium), 2004.

XMLNS

Tim Bray, Dave Hollander, Andrew Layman, editors. *Namespaces in XML*. W3C (World Wide Web Consortium), 1999.

UNICODE

The Unicode Standard, Version 3.2. The Unicode Consortium, 2000.

TWOC

A description of two's complement at Wikipedia:
http://en.wikipedia.org/wiki/Two's_complement