International Standard

ISO/IEC 23415

Information technology — Data Format Description Language (DFDL) v1.0 Specification

First edition
2024-04

Reference number
ISO/IEC 23415:2024(en)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

1 Introduction 9
1.1 Why is DFDL Needed? 10
1.2 What is DFDL? 10
1.2.1 Simple Example 10
1.3 What DFDL is not 13
1.4 Scope of version 1.0 13
2 Overview of the Specification 15
3 Notational and Definitional Conventions 16
3.1 Glossary and Terminology 16
3.2 Failure Types 16
4 The DFDL Information Set (Infoset) 17
4.1 "No Value" 18
4.2 Information Items 18
4.2.1 Document Information Item 18
4.2.2 Element Information Items 18
4.3 DFDL Information Item Order 19
4.4 DFDL Augmented Infoset 19
5 DFDL Schema Component Model 20
5.1 DFDL Simple Types 20
5.2 DFDL Subset of XML Schema 21
5.3 XSD Facets, min/maxOccurs, default, and fixed 22
5.3.1 MinOccurs, MaxOccurs 23
5.3.2 MinLength, MaxLength 23
5.3.3 MaxInclusive, MaxExclusive, MinExclusive, MinInclusive, TotalDigits, FractionDigits 23
5.3.4 Pattern 23
5.3.5 Enumeration Values 23
5.3.6 Default 23
5.3.7 Fixed 24
5.4 Compatibility with Other Annotation Language Schemas 24
6 DFDL Syntax Basics 25
6.1 Namespaces 25
6.2 The DFDL Annotation Elements 25
6.3 DFDL Properties 26
6.3.1 DFDL String Literals 28
6.3.2 DFDL Expressions 32
6.3.3 DFDL Regular Expressions 32
6.3.4 Enumerations in DFDL 32
7 Syntax of DFDL Annotation Elements 33
7.1 Component Format Annotations 33
7.1.1 Property Binding Syntax 34
7.1.2 Empty String as a Representation Property Value 35
7.2 dfdl:defineFormat - Reusable Data Format Definitions 36
7.2.1 Using/Referencing a Named Format Definition: The dfdl:ref Property 36
7.2.2 Inheritance for dfdl:defineFormat 36
7.3 The dfdl:defineEscapeScheme Defining Annotation Element 37
7.3.1 Using/Referencing a Named escapeScheme Definition 37
7.4 The dfdl:escapeScheme Annotation Element 37

7.5 The dfdl:assert Statement Annotation Element ... 38
7.5.1 Properties for dfdl:assert ... 38
7.6 The dfdl:discriminator Statement Annotation Element ... 40
7.6.1 Properties for dfdl:discriminator ... 40
7.7 DFDL Variable Annotations ... 43
7.7.1 dfdl:defineVariable Annotation Element ... 43
7.7.2 The dfdl:newVariableInstance Statement Annotation Element ... 44
7.7.3 The dfdl:setVariable Statement Annotation Element ... 45
8 Property Scoping and DFDL Schema Checking ... 47
8.1 Property Scoping ... 47
8.1.1 Property Scoping Rules ... 47
8.1.2 Providing Defaults for DFDL properties ... 47
8.1.3 Combining DFDL Representation Properties from a dfdl:defineFormat ... 48
8.1.4 Combining DFDL Properties from References ... 48
8.2 DFDL Schema Checking ... 51
8.2.1 Schema Component Constraint: Unique Particle Attribution ... 51
8.2.2 Optional Checks and Warnings ... 51
9 DFDL Processing Introduction ... 53
9.1 Parser Overview ... 53
9.1.1 Points of Uncertainty ... 53
9.1.2 Processing Error ... 54
9.1.3 Recoverable Error ... 54
9.2 DFDL Data Syntax Grammar ... 54
9.2.1 Nil Representation ... 56
9.2.2 Empty Representation ... 56
9.2.3 Normal Representation ... 57
9.2.4 Absent Representation ... 57
9.2.5 Zero-length Representation ... 57
9.2.6 Missing ... 57
9.2.7 Examples of Missing and Empty Representation ... 58
9.2.8 Round Trip Ambiguities ... 58
9.3 Parsing Algorithm ... 58
9.3.1 Known-to-exist and Known-not-to-exist ... 59
9.3.2 Establishing Representation ... 60
9.3.3 Resolving Points of Uncertainty ... 61
9.4 Element Defaults ... 62
9.4.1 Definitions ... 62
9.4.2 Element Defaults When Parsing ... 62
9.4.3 Element Defaults When Unparsing ... 64
9.5 Evaluation Order for Statement Annotations ... 65
9.5.1 Asserts and Discriminators with testKind 'expression' ... 66
9.5.2 Discriminators with testKind 'expression' ... 66
9.5.3 Elements and setVariable ... 66
9.5.4 Controlling the Order of Statement Evaluation ... 66
9.6 Validation ... 66
9.7 Unparser Infoset Augmentation Algorithm ... 67
10 Overview: Representation Properties and their Format Semantics ... 68

11 Properties Common to both Content and Framing 69
11.1 Unicode Byte Order Mark (BOM) 71
11.2 Character Encoding and Decoding Errors 71
11.2.1 Property dfdl:encodingErrorPolicy 72
11.2.2 Unicode UTF-16 Decoding/Encoding Non-Errors 73
11.2.3 Preserving Data Containing Decoding Errors 73
11.3 Byte Order and Bit Order 73
11.4 dfdl:bitOrder Example 73
11.4.1 Example Using Right-to-Left Display for 'leastSignificantBitFirst' 74
11.4.2 dfdl:bitOrder and Grammar Regions 74
12 Framing 75
12.1 Aligned Data 75
12.1.1 Implicit Alignment 76
12.1.2 Mandatory Alignment for Textual Data 76
12.1.3 Mandatory Alignment for Packed Decimal Data 77
12.1.4 Example: AlignmentFill 77
12.2 Properties for Specifying Delimiters 78
12.3 Properties for Specifying Lengths 81
12.3.1 dfdl:lengthKind 'explicit' 82
12.3.2 dfdl:lengthKind 'delimited' 82
12.3.3 dfdl:lengthKind 'implicit' 83
12.3.4 dfdl:lengthKind 'prefixed' 84
12.3.5 dfdl:lengthKind 'pattern' 87
12.3.6 dfdl:lengthKind 'endOfParent' 87
12.3.7 Elements of Specified Length 89
13 Simple Types 92
13.1 Properties Common to All Simple Types 92
13.2 Properties Common to All Simple Types with Text representation 93
13.2.1 The dfdl:escapeScheme Properties 94
13.3 Properties for Bidirectional support for All Simple Types with Text representation 97
13.4 Properties Specific to String 97
13.5 Properties Specific to Number with Text or Binary Representation 99
13.6 Properties Specific to Number with Text Representation 99
13.6.1 The dfdl:textNumberPattern Property 106
13.6.2 Converting logical numbers to/from text representation 110
13.7 Properties Specific to Number with Binary Representation 112
13.7.1 Converting Logical Numbers to/from Binary Representation 113
13.8 Properties Specific to Float/Double with Binary Representation 118
13.9 Properties Specific to Boolean with Text Representation 118
13.10 Properties Specific to Boolean with Binary Representation 119
13.11 Properties Specific to Calendar with Text or Binary Representation 120
13.11.1 The dfdl:calendarPattern property 122
13.11.2 The dfdl:calendarCheckPolicy Property 125
13.12 Properties Specific to Calendar with Text Representation 125
13.13 Properties Specific to Calendar with Binary Representation 126
13.14 Properties Specific to Opaque Types (xs:hexBinary) 127
13.15 Nil Value Processing 127
13.16 Properties for Nillable Elements 127

14 Sequence Groups ... 131
14.1 Empty Sequences ... 131
14.2 Sequence Groups with Separators ... 132
14.2.1 Separators and Suppression ... 133
14.2.2 Parsing Sequence Groups with Separators ... 134
14.2.3 Unparsing Sequence Groups with Separators ... 136
14.3 Unordered Sequence Groups ... 138
14.3.1 Restrictions for Unordered Sequences ... 138
14.3.2 Parsing an Unordered Sequence ... 138
14.3.3 Unparsing an Unordered Sequence ... 140
14.4 Floating Elements ... 140
14.5 Hidden Groups ... 141
15 Choice Groups ... 143
15.1 Resolving Choices ... 144
15.1.1 Resolving Choices via Speculation ... 144
15.1.2 Resolving Choices via Direct Dispatch ... 145
15.1.3 Unparsing Choices ... 145
16 Properties for Array Elements and Optional Elements ... 146
16.1 The dfdl:occursCountKind property ... 146
16.1.1 dfdl:occursCountKind 'fixed' ... 146
16.1.2 dfdl:occursCountKind 'implicit' ... 147
16.1.3 dfdl:occursCountKind 'parsed' ... 147
16.1.4 dfdl:occursCountKind 'expression' ... 147
16.1.5 dfdl:occursCountKind 'stopValue' ... 147
16.2 Default Values for Arrays ... 148
16.3 Arrays with DFDL Expressions ... 148
16.4 Points of Uncertainty ... 148
16.5 Arrays and Sequences ... 148
16.6 Forward Progress Requirement ... 148
16.7 Parsing Occurrences with Non-Normal Representation ... 149
16.8 Sparse Arrays ... 149
17 Calculated Value Properties ... 150
17.1 Example: 2d Nested Array ... 151
17.2 Example: Three-Byte Date ... 151
18 DFDL Expression Language ... 154
18.1 Expression Language Data Model ... 154
18.2 Variables ... 155
18.2.1 Rewinding of Variable Memory State ... 155
18.2.2 Variable Memory State Transitions ... 155
18.3 General Syntax ... 156
18.4 DFDL Expression Syntax ... 157
18.5 Constructors, Functions and Operators ... 158
18.5.1 Constructor Functions for XML Schema Built-in Types ... 158
18.5.2 Standard XPath Functions ... 159
18.5.3 DFDL Functions ... 162
18.5.4 DFDL Constructor Functions ... 164
18.5.5 Miscellaneous Functions ... 165
18.6 Unparsing and Circular Expression Deadlock Errors ... 166

19 DFDL Regular Expressions 167
20 External Control of the DFDL Processor 168
21 Built-in Specifications 169
22 Conformance 170
23 Optional DFDL Features 171
24 Security Considerations 173
25 Authors and Contributors 174
26 Intellectual Property Statement 175
27 Disclaimer 176
28 Full Copyright Notice 177
29 References 178
30 Appendix A: Escape Scheme Use Cases 181
30.1 Escape Character Same as dfdl:escapeEscapeCharacter 181
30.2 Escape Character Different from dfdl:escapeEscapeCharacter 181
30.2.1 Example 1 - Separator ';' 181
30.2.2 Example 2 - Separator 'sep' 182
30.3 Escape Block with Different Start and End Characters 182
30.4 Escape Block with Same Start and End Characters 183
31 Appendix B: Rationale for Single-Assignment Variables 185
32 Appendix C: Processing of DFDL String literals 186
32.1 Interpreting a DFDL String Literal 186
32.2 Recognizing a DFDL String Literal 186
32.3 Recognizing DFDL String Literal Part 186
33 Appendix D: DFDL Standard Encodings 188
33.1 Purpose 188
33.2 Conventions 188
33.3 Specification Template 188
33.4 Encoding X-DFDL-US-ASCII-7-BIT-PACKED 188
33.4.1 Name 188
33.4.2 Translation table 188
33.4.3 Width 188
33.4.4 Alignment 189
33.4.5 Byte Order 189
33.4.6 Example 1 189
33.4.7 Example 2 189
33.5 Encoding X-DFDL-US-ASCII-6-BIT-PACKED 191
33.5.1 Name 191
33.5.2 Translation Table 191
33.5.3 Width 192
33.5.4 Alignment 192
33.5.5 ByteOrder 192
33.5.6 Example 1 192
33.6 References for Appendix D 193
34 Appendix E: Glossary of Terms 194
35 Appendix F: Specific Errors Classified 200
36 Appendix G: Property Precedence 202
36.1 Parsing 202
36.1.1 dfdl:element (simple) and dfdl:simpleType 202

36.1.2 dfdl:element (complex) 205
36.1.3 dfdl:sequence and dfdl:group (when reference is to a sequence) 206
36.1.4 dfdl:choice and dfdl:group (when reference is to a choice) 207
36.2 Unparsing 208
36.2.1 dfdl:element (simple) and dfdl:simpleType 208
36.2.2 dfdl:element (complex) 212
36.2.3 dfdl:sequence and dfdl:group (when reference is a sequence) 213
36.2.4 dfdl:choice and dfdl:group (when reference is a choice) 213

1 Introduction

Data interchange is critically important for most computing. Grid computing, Cloud computing, and all forms of distributed computing require distributed software and hardware resources to work together. Inevitably, these resources read and write data in a variety of formats. General tools for data interchange are essential to solving such problems. Scalable and High-Performance Computing (HPC) applications require high-performance data handling, so data interchange standards must enable efficient representation of data. Data Format Description Language (DFDL) enables powerful data interchange and very high-performance data handling.

One can envisage three dominant kinds of data in the future, as follows:

1. Textual data defined by a format specific schema such as XML[XML] or JSON[JSON].
2. Binary data in standard formats.
3. Data with DFDL descriptors.

Textual XML and JSON data are the most successful data interchange standards to date. All such data are by definition new, meaning created in the Internet era. Because of the large overhead that textual tagging imposes, there is often a need to compress and decompress XML and JSON data. However, there is a high cost for compression and decompression that is unacceptable to some applications. Standardized binary data formats are also relatively new and are suitable for larger data because of the reduced costs of encoding and more compact size. Examples of standard binary formats are data described by modern versions of ASN.1[1] [ASN1], XDR [XDR], Thrift [Thrift], Avro [AVRO], and Google Protocol Buffers [GPB]. These techniques lack the self-describing nature of XML or JSON data. Scientific formats, such as NetCDF[NetCDF] and HDF[HDF] are used by some communities to provide self-describing binary data. There are also standardized binary-encoded XML data formats such as EXI [EXI].

It is an important observation that both XML format and standardized binary formats are *prescriptive* in that they specify or prescribe a representation of the data. To use these, applications must be written to conform to their encodings and mechanisms of expression.

DFDL suggests an entirely different scheme. The approach is *descriptive* in that one chooses an appropriate data representation for an application based on its needs and one then describes the format using DFDL so that multiple programs can directly interchange the described data. DFDL descriptions can be provided by the creator of the format or developed as needed by third parties intending to use the format. That is, DFDL is not a format for data; it is a way of describing any data format[2]. DFDL is intended for data commonly found in scientific and numeric computations, as well as record-oriented representations found in commercial data processing.

DFDL can be used to describe legacy data files, to simplify transfer of data across domains without requiring global standard formats, or to allow third-party tools to easily access multiple formats. DFDL can also be a powerful tool for supporting backward compatibility as formats evolve.

DFDL is designed to provide flexibility and permit implementations that achieve very high levels of performance. DFDL descriptions are separable and native applications do not need to use DFDL libraries to parse their data formats. DFDL parsers can also be highly efficient. The DFDL language is designed to permit implementations that use lazy evaluation of formats and to support seekable, random access to data. The following goals can be achieved by DFDL implementations:

- Density. Fewest bytes to represent information (without resorting to compression). Fastest possible I/O.
- Optimized I/O. Applications can write data aligned to byte, word, or even page boundaries and to use memory mapped I/O to ensure access to data with the smallest number of machine cycles for common use cases without sacrificing general access.

DFDL can describe the same types of abstract data that other binary or textual data formats can describe and, furthermore, it can describe almost any possible representation scheme for those data. It is the intent of DFDL to support canonical data descriptions that correspond closely to the original in-memory representation of the data, and to provide sufficient information to write as well as to read the given format.

[1] ASN.1 with any of the prescribed encoding rules: Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), Canonical Encoding Rules (CER) [ASN1CER] or Packed Encoding Rules (PER) [ASN1PER]

[2] Additional examples of descriptive approaches: ASN1 Encoding Control Notation (also known as ITU-T X.692) [ASN1ECN], BFD: Binary Format Description (BFD) Language [BFD]. The largest set of examples of descriptive approaches are all the various proprietary ad-hoc format description languages found almost universally in every commercial database, analytical, or enterprise software system that must take in data.

1.1 Why is DFDL Needed?

In an era when there are so many standard data formats available the question arises of why DFDL is needed. Ultimately, it is because data formats are rarely a primary consideration when programs are initially created.

Programs are very often written speculatively, that is, without any advance understanding of how important they will become. Given this situation, little effort is expended on data formats since it remains easier to program the I/O in the most straightforward way possible with the programming tools in use. Even something as simple as using an XML-based data format is often harder than just using the native I/O libraries of a programming language.

In time, however, if a software program becomes important either because many people are using it, or it has become important for business or organizational needs, it is often too late to go back and change the data formats. For example, there may be real or perceived business costs to delaying the deployment of a program for a rewrite just to change the data formats, particularly if such rewriting will reduce the performance of the program and increase the costs of deployment.

Indeed, the need for data format standardization for interchange with other software may not be clear at the point where a program first becomes important. Eventually, however, the need for data interchange with the program becomes apparent.

There are, of course, efforts to smoothly integrate standardized data-format handling into programming languages. However, the above phenomena are not going away any time soon and there is a critical role for DFDL since it allows after-the-fact description of evolving data formats.

1.2 What is DFDL?

DFDL is a language for describing data formats. A DFDL description enables *parsing*, that is, it allows data to be read from its native format and presented as a data structure called the *DFDL Information Set or DFDL Infoset*. This information set describes the common characteristics of parsed data that are required of all DFDL implementations and it is fully defined in Section 4. DFDL implementations MAY provide API access to the Infoset as well as conversion of the Infoset into concrete representations such as XML text, binary XML [EXI] , or JSON [JSON]. DFDL also enables *unparsing*[3]*, that is,* allows data to be taken from an instance of a DFDL information set and written out to its native format.

DFDL achieves this by leveraging W3C XML Schema Definition Language (XSD) 1.0. [XSD]

An XML schema is written for the logical model of the data. The schema is augmented with special DFDL annotations and the annotated schema is called a *DFDL Schema*. The annotations are used to describe the native representation of the data.

This approach of extending XSD with format annotations has been extensively used in commercial systems that predate DFDL. The contribution of DFDL for data parsing is creation of a standard for these annotations that is open, comprehensive, and vendor neutral. For unparsing DFDL does more to advance the state of the art by providing some capabilities to automatically compute fields that depend on the length or presence of other data. Prior-generation data format technologies left this difficult task up to application logic to compute.

1.2.1 Simple Example

Consider the following XML data:

```
<w>5</w>
<x>7839372</x>
<y>8.6E-200</y>
<z>-7.1E8</z>
```

The logical model for this data can be described by the following fragment of an XML schema document that simply provides a description of the name and type of each element:

```
  <xs:complexType name="example1">
    <xs:sequence>
      <xs:element name="w" type="xs:int"/>
      <xs:element name="x" type="xs:int"/>
      <xs:element name="y" type="xs:double"/>
      <xs:element name="z" type="xs:float"/>
    </xs:sequence>
```

[3] DFDL uses the term ‘unparsing’ for symmetry with parsing. This is roughly equivalent to the terms ‘marshalling’ or ‘serialization’, but those terms both connote a sequencing order that DFDL does not impose for all formats, so DFDL usesits own distinct term.

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

Now, suppose the same data is represented in a non-XML format. A binary representation of the data can be visualized like this (shown as hexadecimal):

```
0000 0005 0077 9e8c
169a 54dd 0a1b 4a3f
ce29 46f6
```

To describe the same information in DFDL, the original XML schema document that described the data model is annotated (on the type definition) as follows:

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="w" type="xs:int">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
                    binaryNumberRep="binary"
                    byteOrder="bigEndian"
                    lengthKind="implicit"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name="x" type="xs:int ">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
                    binaryNumberRep="binary"
                    byteOrder="bigEndian"
                    lengthKind="implicit"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name="y" type="xs:double">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
                    binaryFloatRep="ieee"
                    byteOrder="bigEndian"
                    lengthKind="implicit"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name="z" type="xs:float" >
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
                    byteOrder="bigEndian"
                    lengthKind="implicit"
                    binaryFloatRep="ieee" />
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```


This simple DFDL annotation expresses that the data are represented in a binary format and that the byte order is big endian. This is all that a DFDL parser needs to read the data.

In the above, there is a standard XML schema annotation structure:

```
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          ...
          ...
        </xs:appinfo>
      </xs:annotation>
```

This encapsulates DFDL *annotation elements*. The source attribute on the xs:appinfo element indicates that the annotation is specifically a DFDL annotation.

Inside the xs:appinfo there is a single DFDL *format annotation*:

```
            <dfdl:element representation="binary"
                   byteOrder="bigEndian"
                   lengthKind="implicit"
                   binaryFloatRep="ieee" />
```

Within the above annotation element, each attribute is a DFDL *property*, and each property-value pair is called a *property binding*. In the above the attribute 'representation' is a DFDL property name. Here the dfdl:element is a DFDL format annotation and the properties in it are generally called DFDL *representation properties*.

Consider if the same data are represented in a text format:

```
5,7839372,8.6E-200,-7.1E8
```

Once again, the same data model can be annotated, this time with properties that provide the character encoding, the field separator (comma) and the decimal separator (period):

```
<xs:complexType>
  <xs:sequence>
    <xs:annotation>
      <xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:sequence encoding="UTF-8" separator="," />
      </xs:appinfo>
    </xs:annotation>
    <xs:element name="w" type="xs:int">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="text"
                        encoding="UTF-8"
                        textNumberRep ="standard"
                        textNumberPattern="####0"
                        textStandardDecimalSeparator="."
                        lengthKind="delimited"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name="x" type="xs:int">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="text"
                        encoding="UTF-8"
                        textNumberRep ="standard"
                        textNumberPattern="#######0"
                        textStandardDecimalSeparator="."
                        lengthKind="delimited"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name="y" type="xs:double">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="text"
                        encoding="UTF-8"
                        textNumberRep ="standard"
                        textNumberPattern="0.0E+000"
                        textStandardDecimalSeparator="."
                        lengthKind="delimited"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
    <xs:element name="z" type="xs:float">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="text"
                        encoding="UTF-8"
                        textNumberRep ="standard"
                        textNumberPattern="0.0E0"
                        textStandardDecimalSeparator="."
                        lengthKind="delimited"/>
        </xs:appinfo>
```

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

Many properties are repeatedly expressed in the example for the sake of simplicity. Later sections of this specification define the mechanisms DFDL provides to avoid this repetition.

1.3 What DFDL is not

DFDL maps data from a native textual or binary representation to an instance of an information set. This can be thought of as a data transformation. However, DFDL is not intended to be a general transformation language and DFDL does not intend to provide a mechanism to map data to arbitrary XML models. There are specific limitations on the data models that DFDL can work to:

1. DFDL uses a subset of XML Schema; in particular, XML attributes cannot be used in the data model.
2. The order of the data in the data model must correspond to the order and structure of the data being described.
3. Recursive definitions are not supported.

Point (2) deserves some elaboration. The XML schema used must be suitable for describing the physical data format. There must be a correspondence between the XML schema's constructs and the physical data structures. For example, generally the elements in the XML schema must match the order of the physical data. DFDL does allow for certain physically unordered formats as well.

The key concept here is that when using DFDL, one does not get to design an XML schema to one's preference and then populate it from data. That would involve two steps: first describing the data format and second describing a transformation for mapping it to the structure of the XML schema. DFDL is only about the format part of this problem. There are other languages, such as XSLT [XSLT], which are for transformation. In DFDL,one describes only the format of the data, and the format constrains the nature of the XML schema one must use in its description.

DFDL is also not intended for describing generic formats like XML or JSON (for which schema-aware parsers exist), nor for prescriptive formats like Google Protocol Buffers [GPB] where the format is never exposed and access is via software libraries.

1.4 Scope of version 1.0

The goals of version 1.0 are as follows:

1. Leverage XML technology and concepts
2. Support very efficient parsers/formatters
3. Avoid features that require unnecessary data copying
4. Support round-tripping, that is, read and write data in a described format from the same description
5. Keep simple cases simple
6. Simple descriptions should be "human readable" to the same degree that XSD is.

The general features of version 1.0 are as follows:

a) Text and binary data parsing and unparsing
b) Validate the data when parsing and unparsing using XSD validation.
c) Defaulted input and output for missing representations
d) Reference – use of the value of a previously read element in subsequent expressions
e) Choice – capability to select among format variations
f) Hidden groups of elements – A description of an intermediate representation the corresponding Infoset items of which are not exposed in the final Infoset.
g) Basic arithmetic in DFDL expressions.
h) Out-of-type value handling (e.g., The string value 'NIL' to indicate nil for an integer)
i) Speculative parsing to resolve uncertainty.
j) Very general parsing capability: Lookahead/Push-back

Version 1.0 of DFDL is a language capable of expressing a wide range of binary and text-based data formats.

DFDL can describe binary data as found in the data structures of COBOL, C, PL1, Fortran, etc., as well as standard binary data in formats like ISO8583 [ISO8583]. DFDL can describe repeating sub-arrays where the length of an array is stored in another location of the structure.

DFDL can describe a wide variety of textual data formats such as HL7, X12, CSV, and SWIFT MT [DFDLSchemas]. Textual data formats often use syntax delimiters, such as initiators, separators and terminators to delimit fields.

DFDL has certain composition properties. I.e., two formats can be nested or concatenated and the combination results in a working format.

The following topics have been deferred to future versions of the standard:

- Extensibility: There are real examples of proprietary data format description languages that were used as the base of experience from which standard DFDL was derived. However, there are no examples of extensible format description languages. Therefore, while extensibility is desirable in DFDL, there is not yet a base of experience with extensibility from which to derive a standard.
- Rich Layering: Some formats require data to be described in multiple passes. Combining these into one DFDL schema requires very rich layering functionality. In these layers one element's value becomes the representation of another element. DFDL V1.0 allows description of only a limited kind of layering.

2 Overview of the Specification

The sections of the specification are

- Section 3, Notational and Definitional Conventions - provides definitions used throughout the specification. Note that terminology is defined at point of first use, but there is a complete Glossary in Appendix E: Glossary of Terms.
- Section 4, The DFDL Information Set (Infoset) - describes the abstract data structure produced by parsing data using a DFDL processor, and which is consumed by a DFDL processor when unparsing data. DFDL contains an expression language, and it is this data structure that the expression language operates on.
- Section 5, DFDL Schema Component Model describes the components that makes up a DFDL schema, and the subset of XML Schema that is used to express them.
- Sections 6, DFDL Syntax Basics and 7, Syntax of DFDL Annotation Elements - describes the syntactic structure of DFDL annotations and introduces the purposes of the various annotations.
- Section 8, Property Scoping and DFDL Schema Checking describes the way DFDL annotations that provide format properties are combined across the parts of the DFDL schema, and also describes static checking that is done on the DFDL schema.
- Section 9, DFDL Processing Introduction covers processing, including the core algorithms for parsing and unparsing data, as well as validation. It introduces the DFDL Data Syntax Grammar, which captures the structure of data that can be described with DFDL, and which is referenced throughout the rest of the specification.
- Section 10, Overview: Representation Properties and their Format Semantics provides an overview of, and Sections 11 to 17 describe in detail, all the DFDL properties. The properties are organized as follows:
    - Common to both Content and Framing (see Section 11)
    - Common Framing, Position, and Length (see Section 12)
    - Simple Type Content (see Section 13) - This is the largest section as it covers properties for all the various simple types, starting with properties that apply to all simple types, then properties for all types with textual representation, and then proceeding through the types, covering textual and binary format properties for each type.
    - Sequence Groups (see Section 14)
    - Choice Groups (see Section 15)
    - Array (i.e., recurring) elements and optional elements (see Section 16)
    - Calculated Values (see Section 17)
- Section 18, DFDL Expression Language covers the XPath-derived expression language that is embedded in DFDL and is used for computing the values of many properties dynamically, as well as for calculated value elements, and assertion checking.
- Section 19, DFDL Regular Expressions, covers the regular expression language used when parsing to isolate elements within the data stream, as well as to check assertions.

The remaining sections and appendices supply additional details of particular importance to implementors of DFDL, or they provide detail and reference material and are referenced from other parts of the specification.

3 Notational and Definitional Conventions

Examples of DFDL schemas provided herein are for illustration purposes only and for clarity they often do not include all the necessary DFDL properties that would be needed for a complete functional DFDL schema.

3.1 Glossary and Terminology

This specification provides definitions of the terms it uses at the point of first use. However, as this specification will not generally be read linearly, but out of order, a complete glossary is provided in Appendix E: Glossary of Terms.

The capitalized key words MUST, MUST NOT, SHALL, SHALL NOT, SHOULD, SHOULD NOT, MAY, REQUIRED, OPTIONAL, and RECOMMENDED in this document are to be interpreted as described in [RFC2119]. Such usage in capital letters is generally about DFDL implementations and their common or distinguishing characteristics.

When describing requirements for correct usage of the DFDL language by a DFDL Schema author, these same words are used, but are not capitalized. For example, the specification may state "The DFDL fillByte property *must* be a single byte or single character." What is intended by "*must*" here is that if the value for that property does not conform, that it is a Schema Definition Error by the schema author.

Similarly, when describing characteristics of data being parsed or being unparsed, and whether that data conforms to the format described by a DFDL schema, these same words may be used. For example,the specification may state "The representation *must* be followed by a terminating delimiter", but what is intended by "*must*" in this case is that the consequence of the data not having that terminating delimiter is a Processing Error because the data does not comply with its format specification.

When describing data, the uncapitalized terms *required* and *optional* in this document have specific formal meanings (introduced in Section 5.3.1,MinOccurs, MaxOccurs) having to do with the way element declarations are annotated in the DFDL language. The data corresponding to such an element declaration is also said to be either required or optional, and the DFDL element declaration is said to be for a required element, or an optional element.

3.2 Failure Types

Where the phrase "MUST be consistent with" is used, it is assumed that a conforming DFDL implementation MUST check for the consistency and issue appropriate diagnostic messages when an inconsistency is found.

There are several kinds of failures that can occur when a DFDL processor is handling data and/or a DFDL schema. These are:

- ***Schema Definition Error*** or ***SDE*** for short - these indicate the DFDL schema is not meaningful. They are generally fatal errors that prevent or stop processing of data.
- ***Processing Error*** - These are errors that occur when parsing or unparsing.
    - At parse time, Processing Errors can cause the parser to search (such as via backtracking) for alternative ways to parse the data as are allowed by the DFDL schema. In that sense parse-time Processing Errors guide the parsing, and when the parser finds an alternative way to parse the data, a prior parse error is said to have been *suppressed*. A parse error that is not suppressed MUST terminate parsing with a diagnostic message.
    - At unparse-time, Processing Errors are generally fatal. They MUST cause unparsing to stop with a diagnostic message.
- ***Validation Error*** - These are errors when optional validation checking is available and enabled. Validation Errors MUST not stop, nor influence, parsing or unparsing behavior. Validation Errors are effectively warnings indicating lack of conformance of the parser output, or the unparser input, with the XML Schema facet constraints, or the XSD maxOccurs and XSD minOccurs values.
- ***Recoverable Error*** - In addition to using XML Schema validation, DFDL also provides the ability to add Recoverable Error assertions to a DFDL schema. These cause diagnostic messages to be created but MUST not stop, nor influence, parsing or unparsing behavior.

4 The DFDL Information Set (Infoset)

This section defines an abstract data set called the ***DFDL Information Set*** (***Infoset***). Its purpose is to define what is provided:

- to an invoking application by a DFDL parser when parsing DFDL-described data using a DFDL Schema;
- to a DFDL unparser by an invoking application when generating DFDL-described data using a DFDL Schema

The DFDL Infoset contains enough information so that a DFDL schema can be defined that enables unparsing the Infoset and reparsing the resultant data stream to produce the same Infoset.

There is no requirement for DFDL-described data to be valid in order to have a DFDL information set.

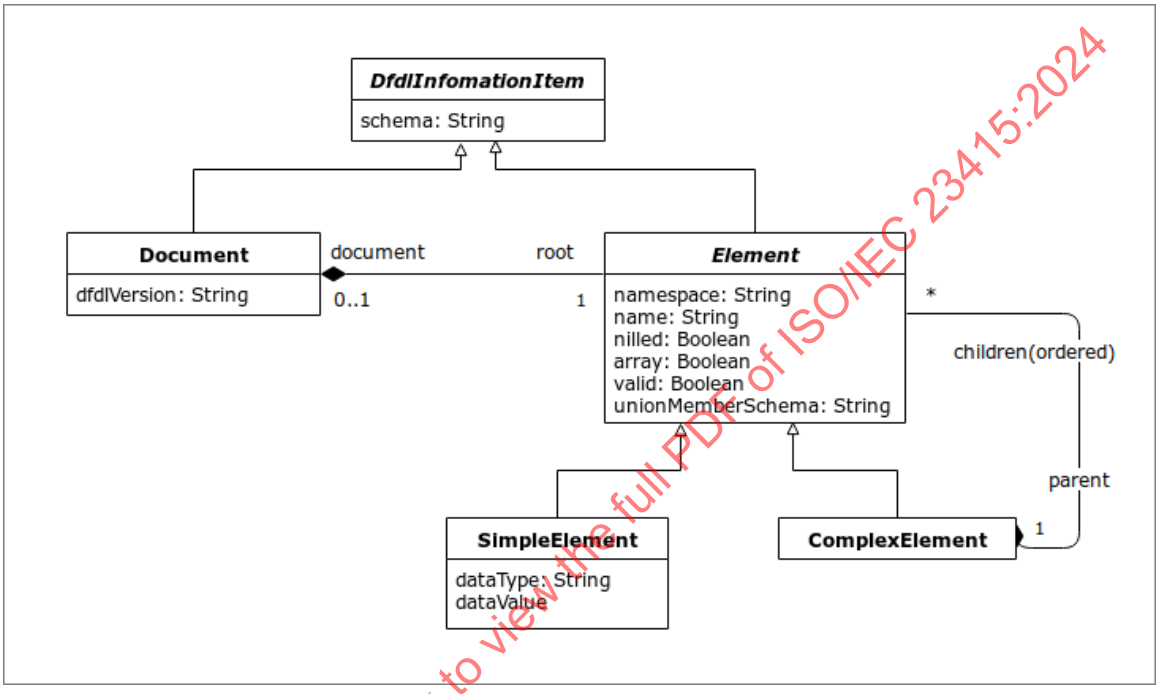

Figure 1 DFDL Infoset Object Model

The DFDL information set is presented above in Figure 1 DFDL Infoset Object Model as an object model using a Unified Modeling Language (UML) class diagram [UML].

The structure of the information set follows the Composite design pattern [Composite]. In case of inconsistency or ambiguity, the following discussion takes precedence.

DFDL describes the format of the physical representation for data whose structure conforms to this model. Note that this model allows hierarchically nested data but does not allow representation of arbitrary connected graphs of data objects.

DFDL information sets may be created by methods (not described in this specification) other than parsing DFDL-described data.

A DFDL information set consists of a number of ***information items***; or just *items* for short. The information set for any well-formed DFDL-described data contains at least a document information item and one element information item. An information item is an abstract description of a part of some DFDL-described data: each information item has a set of associated named ***members***. In this specification, the member names are shown in square brackets, **[thus]**. The types of information item are listed in Section 4.2 Information Items.

The DFDL Information Set does not require or favor a specific implementation interface paradigm. This specification presents the information set as a modified tree for the sake of clarity and simplicity, but there is no requirement that the DFDL Information Set be made available through a tree structure; other types of interfaces, including (but not limited to) event-based and query-based interfaces, are also capable of providing information conforming to the DFDL Information Set.

The terms "information set" and "information item" are similar in meaning to the generic terms "tree" and "node", as they are used in computing. However, the former terms are used in this specification to reduce possible confusion with other specific data models.

The DFDL Information Set is similar in purpose to the XML Information Set [XMLInfoset], however, it is not identical, nor a perfect subset, as there are important differences such as that the DFDL Infoset does not have 'text' nodes that are a primary feature of the XML Infoset, as well as that the contents of strings is much less restricted in the DFDL Infoset.

The DFDL Information Set does not have any specific support for comments. When a data format allows for textual data mixed with a comment syntax, then both that data and the content of the comments correspond to DFDL Information Items.

4.1 "No Value"

In the discussion of Information Items and their members below, some members may sometimes have the value ***no value***, and it is said that such a member has no value. This value is distinct from all other values. In particular it is distinct from the empty string, the empty set, and the empty list, each of which simply has no members. The concept of no-value is also orthogonal to how nillable elements are represented in the Infoset, which uses a separate **[nilled]** boolean flag, not a distinguished value.

4.2 Information Items

An information set contains two different types of information items, as explained in the following sections. Every information item has members. For ease of reference, each member is given a name, indicated **[thus]**.

4.2.1 Document Information Item

There is exactly one ***document information item*** in the information set, and all other information items are accessible through the [root] member of the document information item.

There is no specific DFDL schema component that corresponds to this item. It is a concrete artifact describing the information set.

The document information item has the following members:

- **[root]** The element information item corresponding to the root element declaration of the DFDL Schema.
- **[dfdlVersion]** String. The version of the DFDL specification to which this information set conforms. For DFDL V1.0 this is 'dfdl-1.0'
- **[schema]** String. This member is reserved for future use.

4.2.2 Element Information Items

There is an ***element information item*** for each value parsed from the non-hidden DFDL-described data. This corresponds to an occurrence of a non-hidden element declaration of simple type in the DFDL Schema and is known as a ***simple element information item***.

There is an ***element information item*** for each explicitly declared structure in the DFDL-described data. This corresponds to an occurrence of an element declaration of complex type in the DFDL Schema and is known as a ***complex element information item***.

In this information set, as in an XML document, an array is just a set of adjacent elements with the same name and namespace.

The [root] member of the document information item corresponds to the root element declaration of a DFDL Schema, and all other element information items are accessible by recursively following its [children] member.

An element information item has the following members:

- **[array]** Boolean. True if the item is an array, meaning that it corresponds to an element having maxOccurs value greater than 1, or 'unbounded'.
- **[children]** An ordered set of zero or more element information items. The order they appear in the set is the order implied by the DFDL Schema. 'Ordered set' is not formally defined here, but two operations are assumed: 'count' gives the number of information items, and 'at (index)' gives the element at ordinal position 'index' starting from 1. In a simple element information item this member has no value. In a document information item this member contains exactly one element information item. If the **[nilled]** member is true, then this member has no value.
- **[dataType]** String. The name of the XML Schema 1.0 built-in simple type to which the value corresponds. DFDL supports a subset of these types listed in Section 5.1 DFDL Simple Types.
- **[dataValue]** member has no value, and for a complex element the **[children]** member has no value. If this member is true, then the Infoset item is said to be nil or nilled.
- **[document]** The document information item representing the DFDL information set that contains this element. This element is empty except in the root element of an information set.
- **[name]** String. The local part of the element name.

- **[namespace]** String. The namespace, if any, of the element. If the element does not belong to a namespace, the value is the empty string.
- **[nilled]** Boolean. True if the nillable item is nil. False if the nillable item is not nil. If the element is not nillable this member has no value. If this member is true then for a simple element the
- **[parent]** The complex element information item which contains this information item in its [children] member. In the root element of an information set this member is empty.
- **[schema]** String. A reference to a schema component associated with this information item, if any. If not empty, the value MUST be an absolute or relative Schema Component Designator [SCD].
- **[unionMemberSchema]**[4] String. For simple element information items, this member contains an SCD reference to the member of the union that matched the value of the element. Empty if validation is not enabled. Empty if the element's type is not a union.
- **[valid]** Boolean[5]. True if the element is valid as determined by a DFDL implementation that performs validation checking. A complex element information item is not valid if any of its **[children]** are not valid. Empty if validation is not enabled.

On unparsing, any non-empty values for the **[valid]** or **[unionMemberSchema]** members are ignored. However, in the augmented Infoset which is built during the unparse operation **[valid]** will have a value, and **[unionMemberSchema]** may have a value.

4.3 DFDL Information Item Order

On parsing and unparsing information items are presented in the order they are defined in the DFDL Schema.

4.4 DFDL Augmented Infoset

When unparsing, one begins with the DFDL schema and conceptually with the logical Infoset. This Infoset can be sparsely populated because the DFDL Schema can describe default values and computations to be done to obtain the values of some elements. As unparsing progresses and fills in these defaultable and calculated elements, these new item values augment the Infoset, that is, make it bigger. The resulting Infoset is called the *augmented Infoset*. The details of this augmentation process are described in Section 9.7 Unparser Infoset Augmentation Algorithm.

[4] Also, to support PSVI [PSVI] construction.

[5] The purpose of this member is to support construction of a W3C standard Post Schema Validation Infoset (PSVI) [PSVI] from a DFDL Infoset.

5 DFDL Schema Component Model

When using DFDL, the format of data is described by means of a *DFDL Schema*.

The DFDL Schema Component Model is shown in conceptual UML in Figure 2.

The shaded boxes have direct corresponding XML Schema syntax and therefore appear in DFDL schema. The unshaded boxes are conceptual classes often used in discussion of DFDL schemas. For example, the ModelGroup class is a generalization of Sequence and Choice classes which are the concrete classes corresponding to xs:sequence and xs:choice constructs of the schema. The class Term is a further generalization encompassing not only ModelGroup, but GroupReference, ElementReference, and ElementDeclaration.

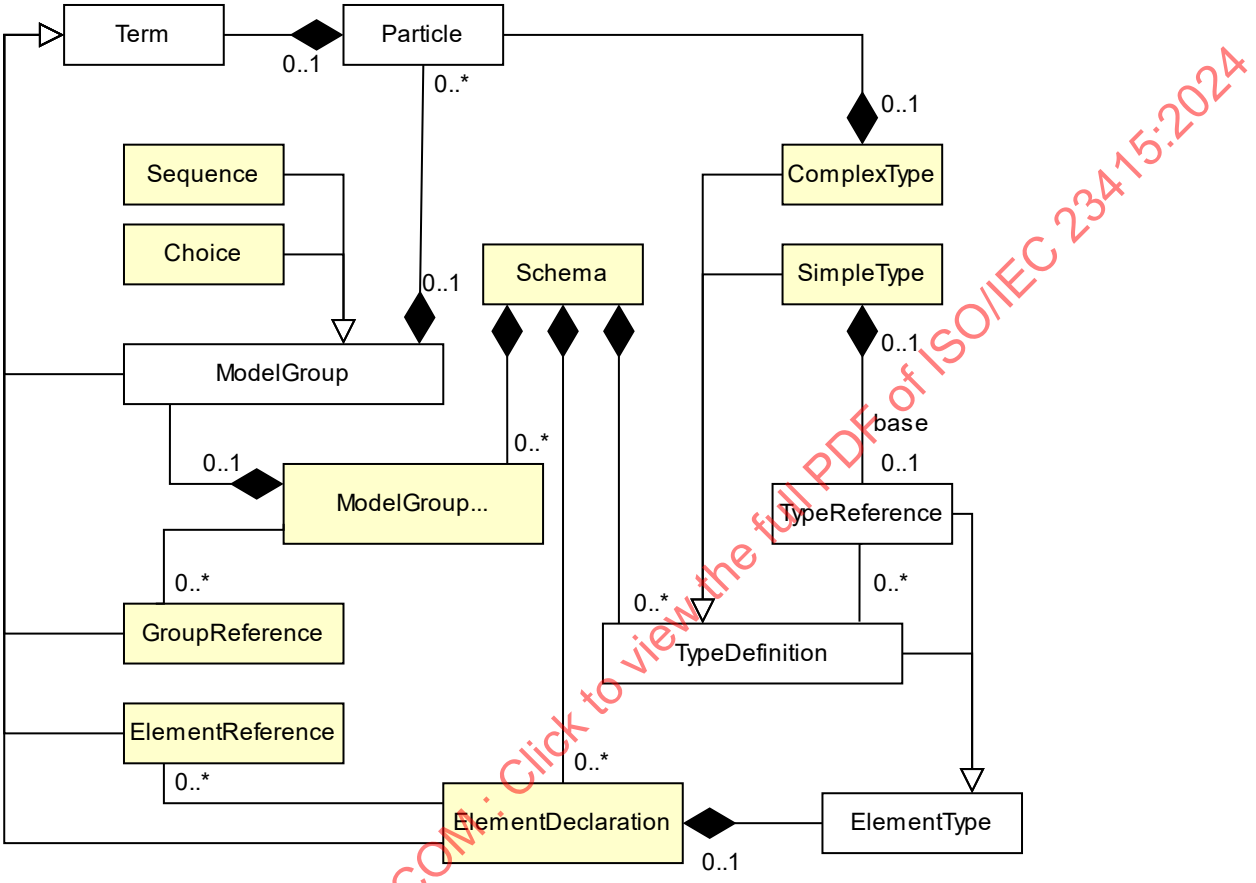

Figure 2 DFDL Schema UML diagram

Each object defined by a class in the above UML is called a *DFDL Schema component*.

The DFDL Schema Model is expressed using a subset of the XML Schema Description Language (XSD). XSD provides a standardized schema language suitable for expressing the DFDL Schema Model.

A DFDL Schema is an XML schema containing only a restricted subset of the constructs available in full W3C XML Schema Description Language. Within this XML schema, special DFDL annotations are distributed that carry the information about the data's format or representation.

A DFDL Schema is a valid XML schema. However, the converse is not true in general since the DFDL Schema Model does not include many concepts that appear in XML schema.

5.1 DFDL Simple Types

The DFDL simple types are shown in Figure 3. The graph shows all the types defined by XML Schema version 1.0, and the subset of these types supported by DFDL are shown as shaded.

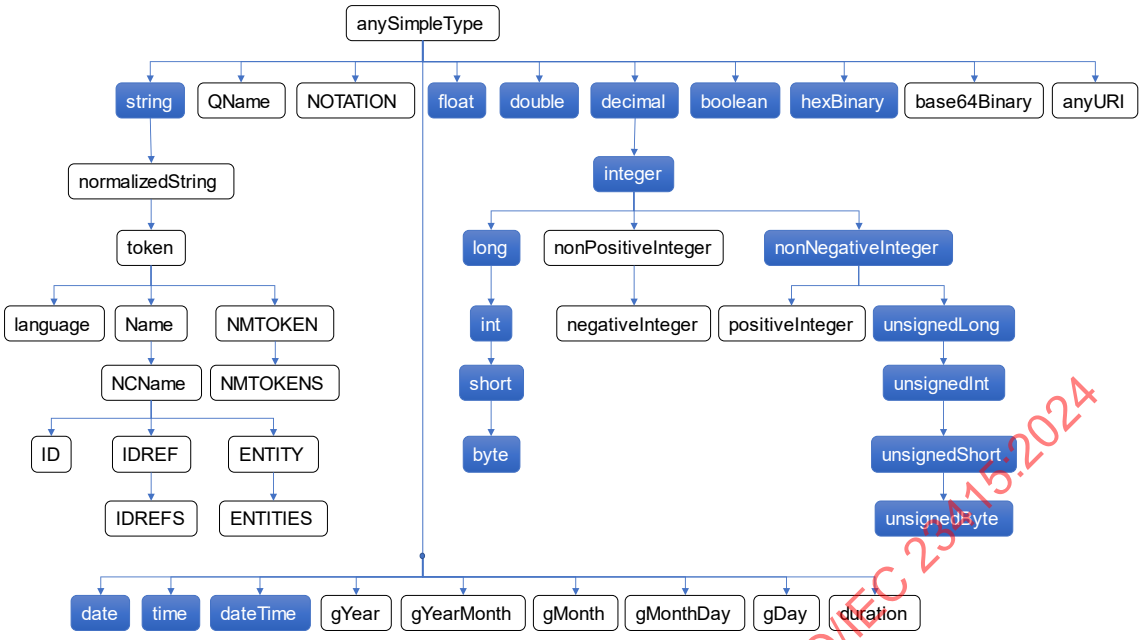

Figure 3 DFDL simple types as a subset of XML Schema types

These types are defined as they are in XML Schema, with the exceptions of:

- String – In DFDL a string can contain any character codes. None are reserved (Including the character with character code U+0000, which is not permitted in XML documents.)

The simple types are placed into logical type groupings as shown in this table:

Logical Type Group	Types
Number	xs:double, xs:float, xs:decimal, xs:integer, xs:nonNegativeInteger, xs:long, xs:int, xs:short, xs:byte, xs:unsignedLong, xs:unsignedInt, xs:unsignedShort, and xs:unsignedByte
String	xs:string
Calendar	xs:dateTime, xs:date, xs:time
Opaque	xs:hexBinary
Boolean	xs:boolean

Table 1: Logical type groupings

Note that DFDL does not have specific types corresponding to time intervals, nor are there special numeric types for geo-coordinates, currency, or complex numbers. These concepts must be described in DFDL using the available types.

5.2 DFDL Subset of XML Schema

The DFDL subset of XSD is a general model for hierarchically nested data. It avoids the XSD features used to describe the peculiarities of XML as a syntactic textual representation of data and avoids features that are simply not needed by DFDL.

The following lists detail the similarities and differences between general XSD and this subset.

DFDL Schemas consist of:

- Standard XSD namespace management
- Standard XSD import and management for multiple file schemas
- Local element declarations with dimensionality via XSD maxOccurs and XSD minOccurs.
- Global element declarations
- Complex type definitions with empty or element-only content models.
- DFDL appinfo annotations describing the data format
- These simple types: string, float, double, decimal, integer, long, int, short, byte, nonNegativeInteger, unsignedLong, unsignedInt, unsignedShort, unsignedByte, boolean, date, time, dateTime, hexBinary

- These facets: minLength, maxLength, minInclusive, maxInclusive, minExclusive, maxExclusive, totalDigits, fractionDigits, enumeration, pattern (for xs:string type only)
- Fixed values
- Default values
- 'sequence' model groups (without XSD minOccurs and XSD maxOccurs or with both XSD minOccurs="1" and XSD maxOccurs="1")
- 'choice' model groups (without XSD minOccurs and XSD maxOccurs or with both XSD minOccurs="1" and XSD maxOccurs="1")
- Simple type derivations derived by restriction from the allowed built-in types
- Reusable Groups: named model group definitions can only contain one model group
- Element references with dimensionality via XSD maxOccurs and XSD minOccurs.
- Group references without dimensionality
- Nillable attribute is "true" (that is, nillable="true" in the element declaration.)
- Appinfo annotations for sources other than DFDL are permitted and ignored
- Unions; the memberTypes must be derived from the same simple type. DFDL annotations are not permitted on union members.6
- XML Entities
- The xs:schema “elementFormDefault” attribute
- The xs:element “form” attribute

Note: xs:nonNegativeInteger is treated as an unsigned xs:integer.

The following constructs from XML Schema are not used as part of the DFDL Schema Model of DFDL v1.0 schemas; however, they are all reserved[7] for future use since the data model may be extended to use them in future versions of DFDL:

- Attribute declarations (local or global)
- Attribute references
- Attribute group definitions
- Complex type derivations where the base type is not xs:anyType.
- Complex types having mixed content models or simple content models
- List simple types
- Union simple types where the member types are not derived from the same simple type.
- These atomic simple types: normalizedString, token, Name, NCName, QName, language, positiveInteger, nonPositiveInteger, negativeInteger, gYear, gYearMonth, gMonth, gMonthDay, gDay, ID, IDREF, IDREFS, ENTITIES, ENTITY, NMTOKEN, NMTOKENS, NOTATION, anyURI, base64Binary
- XSD maxOccurs and XSD minOccurs on model groups (except if both are '1')
- XSD minOccurs = ‘0’ on branches of xs:choice model groups
- Identity Constraints
- Substitution Groups
- xs:all groups
- xs:any element wildcards
- Redefine - This version of DFDL does not support xs:redefine. DFDL schemas must not contain xs:redefine directly or indirectly in schemas they import or include.
- whitespace facet
- Recursively defined types and elements (defined by way of type, group, or element references)

5.3 XSD Facets, min/maxOccurs, default, and fixed

XSD element declarations and references can carry several properties that express constraints on the described data. These constraints are mainly used for validation. These properties include:

- the facets
- minOccurs, maxOccurs
- default
- fixed

[6] The purpose of unions is to allow multiple constraints via facets such as multiple independent range restrictions on numbers. This enhances the ability to do rich validation of data.

[7] By reserved it is intended that conforming DFDL v1.0 implementations MUST NOT assign semantics to them.

The facets and the types they are applicable to are:

- minLength maxLength (for types xs:string, and xs:hexBinary)
- pattern
- enumeration (all types except xs:boolean)
- maxInclusive, maxExclusive, minExclusive, minInclusive (for Number and Calendar types in Section 5.1)
- totalDigits (for type xs:decimal and all supported integer types descending from xs:decimal in Section 5.1)
- fractionDigits (for type xs:decimal)

The facets (but not XSD maxOccurs nor XSD minOccurs) are also checked by the dfdl:checkConstraints DFDL expression language function.

The following sections describe these in more detail.

5.3.1 MinOccurs, MaxOccurs

XSD minOccurs and XSD maxOccurs are used in these definitions:

- An element declaration or reference where XSD minOccurs is greater than zero is said to be a required element.
- An element declaration or reference where XSD minOccurs is equal to zero is said to be an optional element.
- A required element or optional element where XSD maxOccurs is greater than 1 is also said to be an array element.

When validating, XSD minOccurs and XSD maxOccurs are used to determine the minimum and maximum valid number of occurrences of an element.

The XSD minOccurs and XSD maxOccurs values are interpreted in conjunction with the DFDL dfdl:occursCountKind property. See Section 16, Properties for Array Elements and Optional Elements, for more details.

5.3.2 MinLength, MaxLength

These facets are used:

- When dfdl:lengthKind is "implicit" and type is xs:string or xs:hexBinary. In that case the length is given by the value of the XSD maxLength facet. In this case the XSD minLength facet is required to be equal to the XSD maxLength facet (Schema Definition Error otherwise).
- For validation of variable length string elements.

5.3.3 MaxInclusive, MaxExclusive, MinExclusive, MinInclusive, TotalDigits, FractionDigits

- Used for validation only

The format of numbers is not derived from these facets. Rather DFDL properties are used to specify the format.

5.3.4 Pattern

- Allowed only on elements of type xs:string or types derived from it in Section 5.1.
- Used for validation only

It is important to avoid confusion of the pattern facet with other uses of regular expressions that are needed in DFDL (for example, to determine the length of an element by regular expression matching).

Note: in XSD, pattern is about the lexical representation of the data, and since all is text there, everything has a lexical representation. In DFDL only strings are guaranteed to have a lexical and logical value that is identical.

5.3.5 Enumeration Values

Enumerations are used to provide a list of valid values in XSD.

- Used for validation only

Note: in DFDL XSD enumerations are not used as a means to define symbolic constants. These may be captured using dfdl:defineVariable constructs so they can be referenced from expressions.

5.3.6 Default

The XSD default property is used both when parsing and unparsing, to provide the default value of an element when the situation warrants it. See 9.4 Element Defaults.

Note that the XSD fixed and XSD default properties are mutually exclusive on an element declaration.

5.3.7 **Fixed**

The XSD fixed property is used in the same ways as the XSD default property but in addition:

- To constrain the logical value of an element when validating.

Note that the XSD fixed and XSD default properties are mutually exclusive on an element declaration.

5.4 Compatibility with Other Annotation Language Schemas

A DFDL Schema only applies DFDL annotations on a subset of the XML Schema constructs. Hence,one normally thinks that a DFDL schema cannot contain any of the constructs outside of the DFDL subset. For example, the DFDL subset of XML Schema does not use attributes, hence, a DFDL schema normally would not contain attribute declarations.

There is an exception to this, however. One reason to xs:include/xs:import another XML schema document is purely for its use in validating annotations within the schema itself. Such an XML schema is describing not data, but a schema language extension of non-DFDL xs:annotation elements to be used in the rest of the schema.

Hence, the complete set of files making up a schema by way of xs:include/xs:import may include a mixture of DFDL schemas that use only the DFDL subset of XSD, as well as other XML Schemas that describe just annotations. These annotation schemas are unrestricted by the DFDL subset of XML Schema. For example, they may include elements containing xs:attribute declarations.

A DFDL processor needs a way to tell these schema files apart so that it can enforce the DFDL subset in schema files that are describing data formats and ignore the XML schema files that are for unknown annotation languages that are to be ignored by the DFDL processor.

Hence, this rule: a DFDL implementation MUST ignore any schema file included or imported by a DFDL schema if the top level xs:schema element of that included/imported schema does not have an XML namespace binding for the DFDL namespace.

6 DFDL Syntax Basics

Using DFDL, a data format is described by placing special annotations at various positions within an XML schema. A DFDL processor requires these annotations, along with the structural information of the enclosing XML schema, to make sense of the physical data model.

6.1 Namespaces

The xs:appinfo source URI http://www.ogf.org/dfdl/ is used to distinguish DFDL annotations from other annotations.

The element and attribute names in the DFDL syntax are in a namespace defined by the URI http://www.ogf.org/dfdl/dfdl-1.0/[8]. All symbols in this namespace are reserved. DFDL implementations MUST NOT provide extensions to the DFDL standard using names in this namespace. Within this specification, the namespace prefix for DFDL is "dfdl" referring to the namespace http://www.ogf.org/dfdl/dfdl-1.0/.

Attributes on DFDL annotations that are not in the DFDL namespace or in no namespace are ignored by a DFDL processor.

A DFDL Schema document contains XML schema annotation elements that define and assign names to parts of the format specification. These names are defined using the target namespace of the schema document where they reside and are referenced using QNames in the usual manner. A DFDL schema document can include or import another schema document, and namespaces work in the usual manner for XML schema documents. The *schema* as a whole includes all additional schema documents referenced through import and include. Generally, in this specification, when referring to the DFDL Schema this is intended to mean the schema as a whole. When referring to a specific document, the term DFDL Schema document is used.

6.2 The DFDL Annotation Elements

DFDL annotations must be positioned specifically where DFDL annotations are allowed within an XML schema document. These positions are known as *annotation points*. When an annotation is positioned at an annotation point, it binds some additional information to the schema component that encloses it. The description of a data format is achieved by correctly placing annotations on the structural components of the schema.

DFDL specifies a collection of annotations for different purposes. They are organized into three different annotation types: Format Annotations, Statement Annotations, and Defining Annotations

At any single annotation point of the schema there can be only one format annotation, but there can be several statement annotations. There are rules about which of these are allowed to co-exist which are described in sections about those specific annotation types.

The *resolved set of annotations* for an annotation point is a combined set of annotations taken from:

1. a simple type definition and the base simple type it references.
2. an element declaration and the type definition from (1) it references.
3. an element reference and the global element declaration from (2) it references.
4. a group reference and the global group definition it references

Annotation Type	Annotation Element	Description
Format Annotation	dfdl:choice	Defines the physical data format properties of an xs:choice group. See Section 7.1.
	dfdl:element	Defines the physical data format properties of an xs:element and xs:element reference. See Section 7.1.
	dfdl:format	Defines the physical data format properties for multiple DFDL schema constructs. Used on an xs:schema and as a child of a dfdl:defineFormat annotation. This includes aspects such as the encodings, separators, and many more. See Section 7.1.
	dfdl:group	Defines the physical data format properties of an xs:group reference. See Section 7.1.
	dfdl:property	Used in the syntax of format annotations. See Section 7.1.1.2.

[8] Note that the trailing slash is required.

	dfdl:sequence	Defines the physical data format properties of an xs:sequence group. See Section 7.1.
	dfdl:simpleType	Defines the physical data format properties of an xs:simpleType. See Section 7.1.
	dfdl:escapeScheme	Defines the scheme by which quotation marks and escape characters can be specified. This is for use with delimited text formats. See Section 7.4.
Statement Annotation	dfdl:assert	Defines a test to be used to ensure the data are well formed. Assert is used only when parsing data. See Section 7.5
	dfdl:discriminator	Defines a test to be used when resolving choice branches and optional element occurrences. A dfdl:discriminator is used only when parsing data. See Section 7.6
	dfdl:newVariableInstance	Creates a new instance of a variable. See Section 7.7.2
	dfdl:setVariable	Sets the value of a variable whose declaration is in scope See Section 7.7.3
Defining Annotation	dfdl:defineEscapeScheme	Defines a named, reusable escapeScheme See Section 7.3
	dfdl:defineFormat	Defines a reusable data format by collecting together other annotations and associating them with a name that can be referenced from elsewhere. See Section 7.2
	dfdl:defineVariable	Defines a variable that can be referenced elsewhere. This can be used to communicate a parameter from one part of processing to another part. See Section 7.7

Table 2 - DFDL Annotation Elements

DFDL defining annotation elements may only appear at *top-level*, that is, as annotation children of the xs:schema element. The order of their appearance does not matter, nor does their position relative to other children of the xs:schema.

6.3 DFDL Properties

A DFDL *property* is a specific DFDL construct that tells the DFDL processor some characteristic about the data format.

Properties carried on the component format annotations (See Section 7.1) are called *format properties*. A format property that is used to describe a physical characteristic of a component is called a *representation* property.

Properties on DFDL annotations may have values of one or more of the following types

- Enumeration
  The property value is an XSD xs:token the value of which is one of the allowed values listed in the property description.

  Example: the dfdl:lengthKind property, which has values taken from “delimited”, “fixed”, “explicit”, “implicit”, “prefixed”, “pattern”, and “endOfParent”. For example:

  ```
  lengthKind='delimited'
  ```

- DFDL string literal (Section 6.3.1):
  The property value represents a sequence of literal bytes or characters which represent data which appears in the data stream. The value type is a restriction of the XSD xs:token that further disallows the space character. DFDL entities must be used to express whitespace in a DFDL String Literal.

  Example: the dfdl:terminator property, which expresses characters or bytes to be found in the data stream to mark the termination of an element or model group instance. An example terminator might be:

  ```
  terminator='%NL;'
  ```

  This uses DFDL’s string-literal character class entity syntax (see Section 6.3.1.3) to express that the element or model group is terminated by a line ending in the data stream.

- DFDL expression (Section 6.3.2)
  The property is an xs:string the value of which is a DFDL expression that returns a value derived from other property values and/or from the DFDL Infoset. Leading and trailing whitespace is trimmed for DFDL expressions.

  Example: the dfdl:occursCount property takes an expression which commonly looks in the Infoset via an expression, to obtain the count from another element. An example dfdl:occursCount property might be:

  ```
  occursCount='{ ../hdr/count }'
  ```

- DFDL regular expression (Section 6.3.3)
  The property is an xs:string the value of which is a regular expression that can be used as a pattern to calculate the length of an element by applying that pattern to the sequence of literal bytes or characters which appear in the data stream. Note that leading and trailing whitespace is not trimmed and is part of the regular expression value.

  Example: the dfdl:lengthPattern property takes a regular expression which is used to scan the data stream for matching data. An example might be:

  ```
  lengthPattern="\w{1,5};"
  ```

  This scans the data stream for from 1 to 5 word-characters followed by a semi-colon character.

- Logical Value.
  The property value is a string that describes a logical value. The type of the logical value is one of the XML schema simple types. The string must conform to the XML schema lexical representation for the type.

  Example: the dfdl:nilValue property can be used to provide a logical value that if it matches the element's logical value is used to indicate the data is nilled. For example for an element of type xs:int:

  ```
  nilValue='0'
  ```

- QName
  The property value is an XML Qualified Name as specified in "Namespaces in XML" [XMLNamespaces]

  Example: The dfdl:escapeSchemeRef property refers to a named escape scheme definition via its qualified name. For example:

  ```
  escapeSchemeRef='ex:backslashScheme'
  ```

Some properties accept a list or union of types

- List of DFDL String Literals or Logical Values
  The property value is a whitespace separated list of the specified type. When parsing, if more than one string literal in the list matches the portion of the data stream being evaluated then the longest matching value in the list must be used. When unparsing, the first value in the list must be used. String literals containing whitespace or string literals representing the empty string must use character class entities in their syntax.

  Example: The dfdl:separator property below indicates that the items of a sequence are separated either by a comma or a tab character.

  ```
  separator=', %HT;'
  ```

- Union of types and expressions.
  The property value is a union of DFDL expression and exactly one of the other types. The expression must resolve to a value of the other type.

  Example: Below are two examples of the dfdl:length property. One uses an expression that resolves to an unsigned integer, the other a literal unsigned integer.

  ```
  length='{ xs:unsignedInt(../hdr/len) }'

  length='14'
  ```

- Union of types.
  The property value is a union of two or more types. The type is often dependent on the value of another property.

For example, dfdl:nilValue can be a List of DFDL String Literals or a List of Logical Values depending on dfdl:nilKind. Another example is the dfdl:alignment property which can have as its value an unsigned integer or the distinguished enum value 'implicit'.

6.3.1 DFDL String Literals

DFDL String Literals represent a sequence of literal bytes or characters which appear in the data stream. This presents the following challenges:

- the literal characters in the data stream might not be in the same character set encoding as the DFDL schema
- it may be necessary to specify a literal character which is not valid in an XML document
- it may be necessary to specify one or more raw byte values

A DFDL string literal can describe any of the following types of literal data in any combination:

- a single literal character in any encoding
- a string of literal characters in any encoding
- one or more characters from a set of related characters (e.g. end-of-line characters)
- a literal byte value

A DFDL string literal is therefore able to describe any arbitrary sequence of bytes and characters.

Details on how a string literal is matched against the data stream for parsing are given in Appendix C: Processing of DFDL String literals.

Empty String: The special DFDL entity %ES; is provided for describing an empty string or an empty byte sequence. The %ES; entity is the only way to do this. A DFDL string literal with value "" (the empty string) is usually invalid. There are a few properties that explicitly allow an empty DFDL String Literal, and these properties assign a property-specific meaning to the empty string value.

Whitespace: When whitespace must be used as part of a property value, the DFDL string literal must use entities (such as %WSP;) to represent the whitespace. (This allows a property to represent lists of DFDL string literals by using literal spaces to separate list elements.)

6.3.1.1 Character strings in DFDL String Literals

A literal string in a DFDL Schema is written in the character set encoding specified by the XML directive that begins all XML documents:

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

In this example, the DFDL schema is written in UTF-8, so any literal strings contained in it, and particularly string literals found in its representation property bindings in the format annotations, are expressed in UTF-8.

However, these strings are being used to describe features of text data that are commonly in other character set encodings. For example,a DFDL schema may describe EBCDIC data that is comma separated. A comma in EBCDIC has a single-byte code unit of 0x6B in the data, the numeric value of which does not correspond to the Unicode character code for comma which is U+002C. However, when the schema indicates that an item is "," (comma) separated and specifies this using a string literal along with specifying the 'encoding' property to be 'ebcdic-cp-us' then this means that the data are separated by EBCDIC commas regardless of what character set encoding is used to write the DFDL Schema.

```
<?xml version="1.0" encoding="UTF-8">
<xs:schema ...>
    ...
    <dfdl:format encoding="ebcdic-cp-us" separator=","/>
    ...
</xs:schema>
```

When a DFDL processor uses the separator expressed in this manner, the string literal "," is *translated* into the character set encoding of the data it is separating as specified by the dfdl:encoding representation property. Hence, in this case the processor would be searching the data for a character with codepoint 0x6B (the EBCDIC comma), not a UTF-8 or Unicode (0x2C) comma which is what exists in the DFDL schema document.

6.3.1.2 DFDL Character Entities, Character Class Entities, and Byte Values in String Literals

DFDL character entities specify a single Unicode character and provide a convenient way to specify code points that appear in the data stream but would be difficult to specify in XML strings. For example, DFDL character entities can express common non-printable characters or code points, such as 0x00, that are not

valid in XML documents. DFDL entities are based on XML entities, which can also be used in a DFDL schema. Examples:

```
separator='%HT;'
terminator='%WSP*;//'
fillByte='%#x00;'
textStringPadCharacter='%#x7F;'
```

In some cases, regular XML character entities may be used instead. For example, the above '%#x7F;' could be expressed as '&#x7F;' but this is not always the case. There is no way in XSD to express the character code 0 (i.e., the ASCII NUL code point), even as an XML character entity; hence, one must often use DFDL character entities like '%#x00;' above, or their named equivalents. The DFDL string literal syntax allows the author to always use DFDL character entity syntax instead of jumping back and forth between XSD character entities and DFDL character entities.

The following grammar gives the syntax of DFDL String Literals generally, including the various kinds of entities.

```
DfdlStringLiteral       ::=  (DfdlStringLiteralPart)+ | DfdlESEntity

DfdlStringLiteralPart   ::=  LiteralString | DfdlCharEntity | DfdlCharClass |
                             ByteValue

LiteralString           ::=  A string of literal characters

DfdlCharEntity          ::=  DfdlEntity | DecimalCodePoint |
                             HexadecimalCodePoint

DfdlCharClass           ::=  '%' DfdlCharClassName ';'

ByteValue               ::=  '%#r' [0-9a-fA-F]{2} ';'

DfdlEntity              ::=  '%' DfdlEntityName ';'

DecimalCodePoint        ::=  '%#' [0-9]+ ';'

HexadecimalCodePoint    ::=  '%#x' [0-9a-fA-F]+ ';'

DfdlEntityName          ::=  'NUL'|'SOH'|'STX'|'ETX'|
                             'EOT'|'ENQ'|'ACK'|'BEL'|
                             'BS'|'HT'|'LF'|'VT'|'FF'|
                             'CR'|'SO'|'SI'|'DLE'|
                             'DC1'|'DC2'|'DC3'|'DC4'|
                             'NAK'|'SYN'|'ETB'|'CAN'|
                             'EM'|'SUB'|'ESC'|'FS'|
                             'GS'|'RS'|'US'|'SP'|
                             'DEL'|'NBSP'|'NEL'|'LS'

DfdlCharClassName       ::=  DfdlNLEntity | DfdlWSPEntity | DfdlWSPStarEntity |
                             DfdlWSPPlusEntity

DfdlNLEntity            ::=  'NL'

DfdlWSPEntity           ::=  'WSP'

DfdlWSPStarEntity       ::=  'WSP*'

DfdlWSPPlusEntity       ::=  'WSP+'

DfdlESEntity            ::=  'ES'
```

Table 3 DFDL Character Entity, Character Class Entity, and Byte Value Entity Syntax

Using %% inserts a single literal "%" into the string literal. This "%" is subject to character set encoding translation as is any other character.

A HexadecimalCodePoint provides a hexadecimal representation of the character's code point in ISO/IEC 10646.

A DecimalCodePoint provides a decimal representation of the character's code point in ISO/IEC 10646.

A DfdlEntityName is one of the mnemonics given in the following tables.

Mnemonic	Meaning	Unicode Character Code
NUL	null	U+0000
SOH	start of heading	U+0001
STX	start of text	U+0002
ETX	end of text	U+0003
EOT	end of transmission	U+0004
ENQ	enquiry	U+0005
ACK	acknowledge	U+0006
BEL	bell	U+0007
BS	backspace	U+0008
HT	horizontal tab	U+0009
LF	line feed	U+000A
VT	vertical tab	U+000B
FF	form feed	U+000C
CR	carriage return	U+000D
SO	shift out	U+000E
SI	shift in	U+000F
DLE	data link escape	U+0010
DC1	device control 1	U+0011
DC2	device control 2	U+0012
DC3	device control 3	U+0013
DC4	device control 4	U+0014
NAK	negative acknowledge	U+0015
SYN	synchronous idle	U+0016
ETB	end of transmission block	U+0017
CAN	cancel	U+0018
EM	end of medium	U+0019
SUB	substitute	U+001A
ESC	escape	U+001B
FS	file separator	U+001C
GS	group separator	U+001D
RS	record separator	U+001E
US	unit separator	U+001F
SP	space	U+0020
DEL	delete	U+007F
NBSP	no break space	U+00A0
NEL	Next line	U+0085
LS	Line separator	U+2028

Table 4 DFDL Entities

6.3.1.3 DFDL Character Class Entities in DFDL String Literals

The following DFDL character classes are provided to specify one or more characters from a set of related characters.

Mnemonic	Meaning	Unicode Character Code(s)
NL	Newline On parse any one of the single characters CR, LF, NEL or LS or the character combination CRLF. On unparse the value of the dfdl:outputNewLine property is output, which must specify one of the single characters %CR;, %LF;, %NEL;, or %LS; or the character combination %CR;%LF;.	U+000A LF U+000D CR U+000D U+000A CRLF U+0085 NEL U+2028 LS
WSP	Single whitespace On parse any whitespace character On unparse a space (U+0020) is output	U+0009-U+000D (Control characters) U+0020 SPACE U+0085 NEL U+00A0 NBSP U+1680 OGHAM SPACE MARK U+180E MONGOLIAN VOWEL SEPARATOR U+2000-U+200A (different sorts of spaces) U+2028 LSP U+2029 PSP U+202F NARROW NBSP U+205F MEDIUM MATHEMATICAL SPACE U+3000 IDEOGRAPHIC SPACE
WSP*	Optional Whitespaces On parse whitespace characters are ignored. On unparse nothing is output	Same as WSP
WSP+	Whitespaces On parse one or more whitespace characters are ignored. It is a Processing Error if no whitespace character is found. On unparse a space (U+0020) is output.	Same as WSP
ES	Empty String Used in whitespace separated lists when empty string is one of the values.	

Table 5 DFDL Character Class Entities

6.3.1.4 DFDL Byte Value Entities in DFDL String Literals

DFDL byte-value entities provide a way to specify a single byte as it appears in the data stream without any character set encoding translation. To specify a string of byte values, a sequence of two or more byte-value entities must be used. The syntax is in Table 3 DFDL Character Entity, Character Class Entity, and Byte Value Entity Syntax above. Example:

```
%#rFF;
```

In this notation the "r" can be thought of as short for "raw", as byte value entities are said to denote "raw bytes".

6.3.2 DFDL Expressions

Some DFDL properties allow DFDL expressions (see Section 18 DFDL Expression Language) to be used so that the property can be set dynamically at processing-time.

The general syntax of expressions is "{" expression "}"

The rules for recognizing DFDL expressions are

- Discard any leading and trailing whitespace.
- Must start with a '{' in the first position and end with '}' in the last position.
- '{' in any position other than the first is treated as a literal.
- '}' in any position other than the last position is treated as a literal.
- '{{' as the first characters are treated as the literal '{' and not as the start of a DFDL expression.

DFDL expressions reference other items in the Infoset or augmented Infoset using absolute or relative paths.

DFDL expressions that are used to provide the value of DFDL properties in the dfdl:format annotation on the top level xs:schema declaration must not contain relative paths.

6.3.3 DFDL Regular Expressions

Some properties expect a regular expression to be specified. The DFDL Regular Expression language is defined in Section 19, DFDL Regular Expressions.

6.3.4 Enumerations in DFDL

Some DFDL properties accept an enumerated list of valid values. It is a Schema Definition Error if a value other than one of the enumerated values is specified. The case of the specified value must match the enumeration. An enumeration is of type string unless otherwise stated.

7 Syntax of DFDL Annotation Elements

This section describes the syntax of each of the DFDL annotation elements along with discussion of their basic meanings.

The DFDL annotation elements are listed in **Table 2 - DFDL Annotation Elements**

7.1 Component Format Annotations

A data format can be 'used' or put into effect for a part of the schema by use of the component format annotation elements.

There are specific annotations for each type of schema component that supports only the representation properties applicable to that component. The table below gives the specific annotation for each schema component.

Schema component	DFDL annotation
xs:choice	dfdl:choice
xs:element	dfdl:element
xs:element reference	dfdl:element
xs:group reference	dfdl:group
xs:schema	dfdl:format
xs:sequence	dfdl:sequence
xs:simpleType	dfdl:simpleType

Table 6 DFDL Component Format Annotations

Below are a few examples followed by sections which describe each kind of annotation element in detail. Here is an example of DFDL component format annotation, specifically use of dfdl:element on an xs:element declaration:

```
<xs:schema ...>
  ...
  <xs:element name="root">
    <xs:annotation>
      <xs:appinfo source="http://www.ogf.org/dfdl/">

        <dfdl:element ref="aBaseConfig"
                      representation="text"
                      encoding="UTF-8"/>

      </xs:appinfo>
    </xs:annotation>
  </xs:element>
  ...
</xs:schema>
```

Note that in the above, the DFDL annotation lives inside this surrounding context of xs:annotation and xs:appinfo elements. This is just the standard XSD way of doing annotations. The source attribute is an identifier that separates different families of appinfo annotations.

Below a dfdl:format annotation is used inside a dfdl:defineFormat annotation to define a named reusable set of format properties that can be referenced from another format annotation.

```
<xs:schema ...>
  ...
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">

      <dfdl:defineFormat name="baseFormat">
        <dfdl:format byteOrder="bigEndian" encoding="ascii"/>
      </dfdl:defineFormat>

    </xs:appinfo>
  </xs:annotation>
  ...
```

```
</xs:schema>
```

A dfdl:format annotation at the top level of a schema, that is as an annotation child element on the xs:schema, provides a set of default properties for the lexically enclosed schema document. (See 8.1.2 Providing Defaults for DFDL properties.)

```
<xs:schema ...>
  ...
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">

        <dfdl:format
           representation="binary"
           byteOrder="bigEndian"
           encoding="ascii"/>

    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

7.1.1 Property Binding Syntax

A *property binding* is the syntax in a DFDL schema that gives a value to a property. Up to this point, the examples in this document have all used a specific syntax for property bindings called *attribute form*. However, the format properties may be specified in any one of three forms:

1. Attribute form
2. Element form
3. Short form

A DFDL property may be specified using any of the forms with the following exceptions:

- The dfdl:ref property may be specified in attribute or short form
- The dfdl:escapeSchemeRef property may be specified in attribute or short form
- The dfdl:hiddenGroupRef property may be specified in attribute or short form
- The dfdl:prefixLengthType property may be specified in attribute or short form
- Short form must not be used on the xs:schema element.

It is a Schema Definition Error if the same property is specified in more than one form. That is, there is no priority ordering where one form takes precedent over another.

7.1.1.1 Property Binding Syntax: Attribute Form

Within the format annotation elements are bindings for properties of the form:

```
 PropertyName="Value"
```

For example:

```
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:format encoding="utf-8" separator="%NL;"/>
    </xs:appinfo>
  </xs:annotation>
```

This is the attribute form of property binding.

7.1.1.2 Property Binding Syntax: Element Form

The representation properties can sometimes have complex syntax, so an element form for individual property bindings is provided to ease syntactic expression difficulties. The annotation element is dfdl:property and it has one attribute 'name' which provides the name of the property.

For example:

```
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:format>
        <dfdl:property name='encoding'>utf-8</dfdl:property>
        <dfdl:property name='separator'>%NL;</dfdl:property>
      </dfdl:format>
```

```
    </xs:appinfo>
  </xs:annotation>
```

Element form is mostly used for properties that themselves contain the quotation mark characters and escape characters so that the property value can be expressed without concerns about confusion with the XSD syntax use of these same characters. XML's CDATA encapsulation can be used to allow malformed XML and mismatched quotes to be easily used as representation property values.

Here is an example where a delimiter has a syntax that overlaps with what XML comments look like. Use of XML's CDATA bracketing makes this less clumsy to express than using XML escape characters:

```
<dfdl:property name='initiator'><[CDATA[<!-- ]]></dfdl:property>
```

7.1.1.3 Property Binding Syntax: Short Form

To save textual clutter, short-form syntax for format annotations is also allowed on xs:element, xs:sequence, xs:choice, xs:group (for group references only), and xs:simpleType schema elements. The xs:schema element cannot carry short-form annotations; attribute form must be used instead. Attributes which are in the namespace 'http://www.ogf.org/dfdl/dfdl-1.0/' and whose local name matches one of the DFDL representation properties are assumed to be equivalent to specific DFDL attribute form annotations.

For example, the two forms below are equivalent in that they describe the same data format. The first is the short form of the second:

```
<xs:element name="elem1">
  <xs:complexType>
    <xs:sequence dfdl:separator="%HT;" >
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="elem2">
  <xs:complexType>
    <xs:sequence>
      <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:sequence separator="%HT;" />
      </xs:appinfo></xs:annotation>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Another example:

```
<xs:sequence dfdl:separator=",">
  <xs:element name="elem1" type="xs:int" maxOccurs="unbounded"
                      dfdl:representation="text"
                      dfdl:textNumberRep="standard"
                      dfdl:initiator="["
                      dfdl:terminator="]"/>

  <xs:element name="elem2" type="xs:int" maxOccurs="unbounded">
    <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:element representation="text"
                    textNumberRep="standard"
                    initiator="["
                    terminator="]"/>
    </xs:appinfo></xs:annotation>
  </xs:element>
</xs:sequence>
```

The above show use of short-form property binding syntax for annotating elements and sequences.

7.1.2 Empty String as a Representation Property Value

DFDL provides no mechanism to un-set a property. Setting a representation property's value to the empty string doesn't remove the value for that property but sets it to the empty string value. This may not be a valid value for certain properties.

For example, in non-delimited text data formats, it is sensible for the separator to be defined to be the empty string. This turns off use of separator delimiters. For many other string-valued properties, it is a Schema

Definition Error to assign them the empty string value. For example, the character set encoding property (dfdl:encoding) cannot be set to the empty string.

7.2 dfdl:defineFormat - Reusable Data Format Definitions

To avoid error-prone redundant expression of properties in DFDL schemas, a collection of DFDL properties can be given a name so that they are reusable by way of a *format reference*.

One or more dfdl:defineFormat annotation elements can appear within the annotation children of the xs:schema element.

Each dfdl:defineFormat has a required name attribute.

The construct creates a named data format definition. The value of the name attribute is of XML type NCName. The format name becomes a member of the schema's target namespace. These names must be unique within the namespace.

If multiple format definitions have the same 'name' attribute, in the same namespace, then it is a Schema Definition Error.

Here is an example of a format definition:

```
<xs:schema ...>
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:defineFormat name="baseFormat" >
        <dfdl:format representation="text"
                     encoding="ascii" />
      </dfdl:defineFormat>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

A dfdl:defineFormat serves only to supply a named definition for a format for reuse from other places. It does not cause any use of the representation properties it contains to describe any actual data.

7.2.1 Using/Referencing a Named Format Definition: The dfdl:ref Property

A named, reusable, dfdl:defineFormat definition is used by referring to its name from a format annotation using the dfdl:ref property. For example, here this annotation reuses the format named 'baseFormat':

```
<dfdl:element ref="baseFormat" encoding="ebcdic-cp-us" />
```

The behavior of this dfdl:element definition is as if all representation properties defined by the named dfdl:defineFormat definition for 'baseFormat' were instead written directly on this dfdl:element annotation; however, these are superseded by any representation properties that are defined here such as the dfdl:encoding property in the example above.

7.2.2 Inheritance for dfdl:defineFormat

A dfdl:defineFormat declaration can inherit from another named format definition by use of the dfdl:ref property of the dfdl:format annotation. This allows a single-inheritance hierarchy that reuses definitions. When one definition extends another in this way, any property definitions contained in its direct elements override those in any inherited definition.

An example format that inherits from a named format definition is:

```
<xs:schema ...>
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:defineFormat name="myConfig" >
        <dfdl:format representation="binary"
                     ref="baseFormat" />
      </dfdl:defineFormat>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

Conceptually, the dfdl:ref inheritance chains can be *flattened* and removed by copying all inherited property bindings and then superseding those for which there is a local binding. Throughout this document the discussion assumes inheritance is fully flattened. That is, all dfdl:ref inheritance is first removed by flattening before any other examination of properties occurs.

It is a Schema Definition Error if use of the dfdl:ref property results in a circular path.

7.3 The dfdl:defineEscapeScheme Defining Annotation Element

One or more dfdl:defineEscapeScheme annotation elements can appear within the annotation children of the xs:schema. The dfdl:defineEscapeScheme elements may only appear as annotation children of the xs:schema.

The order of their appearance does not matter, nor does their position relative to other annotation or non-annotation children of the xs:schema.

Each dfdl:defineEscapeScheme has a required name attribute and a required dfdl:escapeScheme child element.

The construct creates a named escape scheme definition. The value of the name attribute is of XML type NCName. The name becomes a member of the schema's target namespace. These names must be unique within the namespace among escape schemes.

If multiple dfdl:defineEscapeScheme definitions have the same 'name' attribute, in the same namespace, then it is a Schema Definition Error.

Each dfdl:defineEscapeScheme annotation element contains a dfdl:escapeScheme annotation element as detailed below.

Here is an example of an escapeScheme definition:

```
<xs:schema ...>
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:defineEscapeScheme name="myEscapeScheme">
        <dfdl:escapeScheme escapeKind="escapeCharacter"
                           escapeCharacter='/' />
        ...
      </dfdl:defineEscapeScheme>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

A dfdl:defineEscapeScheme serves only to supply a named definition for a dfdl:escapeScheme for reuse from other places. It does not cause any use of the representation properties it contains to describe any actual data.

7.3.1 Using/Referencing a Named escapeScheme Definition

A named, reusable, escape scheme is used by referring to its name from a dfdl:escapeSchemeRef property on an element. For example:

```
<xs:element name="foo" type="xs:string" >
  <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
    <dfdl:element representation="text"
                  escapeSchemeRef="myEscapeScheme"/>
  </xs:appinfo></xs:annotation>
</xs:element>
```

7.4 The dfdl:escapeScheme Annotation Element

The dfdl:escapeScheme annotation is used within a dfdl:defineEscapeScheme annotation to group the properties of an escape scheme and allows a common set of properties to be defined that can be reused.

An escape scheme defines the properties that describe the text escaping rules in force when data such as text delimiters are present in the data. There are two variants on such schemes,

- The use of a single escape character to cause the next character to be interpreted literally. The escape character itself is escaped by the escape-escape character.
- The use of a pair of escape strings to cause the enclosed group of characters to be interpreted literally. The ending escape string is escaped by the escape-escape character.

On parsing, the escape scheme is applied after pad characters are trimmed and on unparsing before pad characters are added.

DFDL does not perform any substitutions for ampersand notations like <.

The properties of dfdl:escapeScheme are defined in Section 13.2.1 The dfdl:escapeScheme Properties.

7.5 The dfdl:assert Statement Annotation Element

The dfdl:assert statement annotation element is used to assert truths about a DFDL model that are used when parsing to ensure that the data are well-formed. They are not used when unparsing.

There is a critical distinction between dfdl:assert checks and XSD validation checks.

The dfdl:assert checks guide parsing and the creation of the DFDL Infoset by causing Processing Errors on failure. Conversely XSD validation inspects the values within the Infoset. Validation failures never affect whether the parser is able to produce a DFDL Infoset.

The dfdl:assert checks are performed even when validation is off.

Examples of dfdl:assert elements are below:

```
<dfdl:assert message="Value is not zero." test="{ ../x eq 0}" />

<dfdl:assert message="Precondition violation." >
        {../x le 0 and ../y ne "-->" and ../y ne "<!—" }
</dfdl:assert>


<dfdl:assert message="Postcondition violation."  testKind='expression'>
     {../x ne "'"}
</dfdl:assert>
```

7.5.1 Properties for dfdl:assert

A dfdl:assert annotation contains a test expression or a test pattern. The dfdl:assert is said to be successful if the test expression evaluates to true or the test pattern returns a non-zero length match, and unsuccessful if the test expression evaluates to false or the test pattern returns a zero length match. An unsuccessful dfdl:assert causes either a Processing Error or a Recoverable Error to be issued, as specified by the failureType property of the dfdl:assert.

The testKind property specifies whether an expression or pattern is used by the dfdl:assert. The expression or pattern can be expressed as an attribute or as a value.

```
<dfdl:assert  test="{test expression}" />

<dfdl:assert>
  {test expression}
</dfdl:assert>
```

It is a Schema Definition Error if a test expression or test pattern is specified in more than one form.

It is a Schema Definition Error if both a test expression and a test pattern are specified.

A dfdl:assert can appear as an annotation on these schema components:

- an xs:element declaration (local or global)
- an xs:element reference
- an xs:group reference
- an xs:sequence
- an xs:choice
- an xs:simpleType definition (local or global)

If the resolved set of statement annotations for a schema component contains multiple dfdl:assert statements, then those with testKind 'pattern' are executed before those with testKind 'expression' (the default). However, within each group the order of execution among them is not specified.

If one of the resolved set of asserts for a schema component is unsuccessful, and the failureType of the assert is ‘processingError’, then no further asserts in the set are executed.

Property Name	Description
testKind	Enum (optional) Valid values are 'expression', 'pattern' Default value is 'expression' Specifies whether a DFDL expression or DFDL regular expression pattern is used in the dfdl:assert.

	Annotation: dfdl:assert
test	DFDL Expression Applies when testKind is 'expression' A DFDL expression that evaluates to true or false. If the expression evaluates to true then parsing continues. If the expression evaluates to false then a Processing Error is raised. Any element referred to by the expression must have already been processed or must be a descendent of this element. If a Processing Error occurs during the evaluation of the test expression then the dfdl:assert also fails. It is a Schema Definition Error if testKind is 'expression' or not specified, and an expression is not supplied by either the value of the dfdl:assert element or the value of the test attribute. Annotation: dfdl:assert
testPattern	DFDL Regular Expression Applies when testKind is 'pattern' A DFDL regular expression that is applied against the data stream starting at the data position corresponding to the beginning of the representation. Consequently, the framing (including any initiator) is visible to the pattern.at the start of the component on which the dfdl:assert is positioned. If the pattern matching of the regular expression reads data that cannot be decoded into characters of the current encoding, then the behavior is controlled by the dfdl:encodingErrorPolicy property. See Section 11.2.1 Property dfdl:encodingErrorPolicy for details. If the length of the match is zero then the dfdl:assert evaluates to false and a Processing Error is raised. If the length of the match is non-zero then the dfdl:assert evaluates to true. If a Processing Error occurs during the evaluation of the test regular expression then the dfdl:assert also fails. It is a Schema Definition Error if testKind is 'pattern', and a pattern is not supplied by either the value of the dfdl:assert element or the value of the testPattern property. It is a Schema Definition Error if there is no value for the dfdl:encoding property in scope. It is a Schema Definition Error if dfdl:leadingSkip is other than 0. It is a Schema Definition Error if the dfdl:alignment is not 1 or 'implicit' Annotation: dfdl:assert
message	String or DFDL Expression Defines text to be used as a diagnostic code or for use in an error message, when the assert is unsuccessful. The DFDL Expression must return type xs:string. Any element referred to by the message expression must have already been processed or must be a descendent of this element. There is special treatment for errors that occur while evaluating the message expression. See below for details. Annotation: dfdl:assert
failureType	Enum (optional) Valid values are 'processingError', 'recoverableError'. Default value is 'processingError'. Specifies the type of failure that occurs when the dfdl:assert is unsuccessful. When 'processingError', a Processing Error is raised. When 'recoverableError', a Recoverable Error is raised. If an error occurs while evaluating the test expression, a Processing Error occurs, not a Recoverable Error. Recoverable Errors do not cause backtracking like Processing Errors. Annotation: dfdl:assert

Table 7 dfdl:assert properties

Example of a dfdl:assert with a message expression:

```
<dfdl:assert message="{ fn:concat('unknown case ', ../data1) }">
{  if (...pred1...) then ...expr1...
   else if (...pred2...) then ...expr2...
   else fn:false()
}
</dfdl:assert>
```

The message specified by the message property is issued only if the dfdl:assert is unsuccessful, that is, the test expression evaluates to false or the test pattern returns a zero-length match. If so, and the message property is an expression, the message expression is evaluated at that time.

If a Processing Error or Schema Definition Error occurs while evaluating the message expression, a Recoverable Error is issued to record this error (containing implementation-dependent content), then processing of the assert continues as if there were no problem and in a manner consistent with the failureType property, but using an implementation-dependent substitute message.

7.6 The dfdl:discriminator Statement Annotation Element

DFDL discriminator statement annotations are used during parsing to:

1. resolve *points of uncertainty* (choices, optional elements, array repetition) that cannot be resolved by speculative parsing. See Section 9.1 Parser Overview.
2. remove ambiguity during speculative parsing
3. improve diagnostic behavior when a DFDL parser encounters malformed data.

Discriminators are not used during unparsing.

A DFDL discriminator may contain a test expression that evaluates to true or false. The discriminator is said to be successful if the test evaluates to true and unsuccessful (or fails) if the test evaluates to false. A discriminator may alternatively contain a test regular expression pattern and the discriminator is successful if the test pattern matches with non-zero length and is unsuccessful (or fails) if there is no match or a zero-length match.

A discriminator determines the existence or non-existence of a schema component in the data stream. If the discriminator is successful, then the component is said to be *known to exist*, and any subsequent errors do not cause backtracking at the nearest point of uncertainty. Details of the behavior of a DFDL parser and the role of discriminators are given in Section 9.3 Parsing Algorithm.

Discriminators can also be used to force a resolution earlier during the parsing of a model group so that subsequent parsing errors are treated as Processing Errors of a known schema component rather than a failure to find that schema component. This may greatly improve the efficiency of DFDL parsing in some implementations, as well as improving the diagnostic information provided by a DFDL parser when given malformed data.

Examples of dfdl:discriminator annotation are below :

```
<dfdl:discriminator>
  { ../recType eq 0 }
</dfdl:discriminator>

<dfdl:discriminator test="{ ../recType eq 0}" />
```

When the discriminator's expression evaluates to "false", then it causes a Processing Error, and the discriminator is said to fail.

7.6.1 Properties for dfdl:discriminator

Within a dfdl:discriminator, the testKind property specifies whether an expression or pattern is used by the dfdl:discriminator. The expression or pattern can be expressed as an attribute or as a value.

```
<dfdl:discriminator test="{test expression}" />

<dfdl:discriminator>
    { test expression }
</dfdl:discriminator>
```

It is a Schema Definition Error if the test expression or test pattern is specified in more than one form.

It is a Schema Definition Error if both a test expression and a test pattern are specified.

A dfdl:discriminator can be an annotation on these schema components:

- an xs:element declaration (local or global)
- an xs:element reference
- an xs:group reference
- an xs:sequence
- an xs:choice
- an xs:simpleType definition (local or global)

The resolved set of statement annotations for a schema component can contain only a single dfdl:discriminator or one or more dfdl:assert annotations, but not both. To clarify: dfdl:assert annotations and dfdl:discriminator annotations are exclusive of each other. It is a Schema Definition Error otherwise.

Property Name	Description
testKind	Enum Valid values are 'expression', 'pattern' Default value is 'expression' Specifies whether a DFDL expression or DFDL regular expression is used in the dfdl:discriminator . Annotation: dfdl:discriminator
test	DFDL Expression Applies when testKind is 'expression' A DFDL expression that evaluates to true or false. If the expression evaluates to true then the discriminator succeeds, and parsing continues. If the expression evaluates to false then the discriminator fails, and a Processing Error is raised. If a Processing Error occurs during the evaluation of the test expression then the discriminator also fails. Any element referred to by the expression must have already been processed or is a descendent of this element. The expression must have been evaluated by the time this element and its descendants have been processed or when a Processing Error occurs when processing this element or its descendants. It is a Schema Definition Error if testKind is 'expression' or not specified, and an expression is not supplied by either the value of the dfdl:discriminator element or the value of the test attribute. Annotation: dfdl:discriminator
testPattern	DFDL Regular Expression Applies when testKind is 'pattern' A DFDL regular expression that is applied against the data stream starting at the data position corresponding to the beginning of the representation. Consequently, the framing (including any initiator) is visible to the pattern.at the start of the component on which the dfdl:discriminator is positioned. If the pattern matching of the regular expression reads data that cannot be decoded into characters of the current encoding, then the behavior is controlled by the dfdl:encodingErrorPolicy property. See Section 11.2.1 Property dfdl:encodingErrorPolicy for details. If the length of the match is zero then the dfdl:discriminator evaluates to false and a Processing Error is raised. If the length of the match is non-zero then the dfdl:discriminator evaluates to true. It is a Schema Definition Error if testKind is 'pattern', and a pattern is not supplied by either the value of the dfdl:discriminator element or the value of the testPattern property. It is a Schema Definition Error if there is no value for the dfdl:encoding property in scope. It is a Schema Definition Error if dfdl:leadingSkip is other than 0. It is a Schema Definition Error if the dfdl:alignment is not 1 or 'implicit'

	Annotation: dfdl:discriminator
message	String or DFDL Expression Defines text to be used as a diagnostic code or for use in an error message, when the discriminator is unsuccessful. The DFDL Expression must return type xs:string. Any element referred to by the message expression must have already been processed or must be a descendent of this element. There is special treatment for errors that occur while evaluating the message expression. See below for details. Annotation: dfdl:discriminator

Table 8 dfdl:discriminator properties

The message specified by the message property is issued only if the discriminator is unsuccessful, that is, the test expression evaluates to false or the test pattern returns a zero-length match. If so, and the message property is an expression, the message expression is evaluated at that time.

If a Processing Error or Schema Definition Error occurs while evaluating the message expression, a Recoverable Error is issued to record this error (containing implementation-dependent content), then processing of the discriminator continues as if there were no problem, but in the case of failure using an implementation-dependent substitute message.

Examples of dfdl:discriminator annotations:

```
<xs:sequence>
  <xs:choice>
    <xs:element  name='branchSimple' >
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:discriminator test='{. eq "a"}'       />
        </xs:appinfo>
      </xs:annotation>
    </xs:element>

    <xs:element name='branchComplex' >
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:discriminator test='{./identifier eq "b"}' />
        </xs:appinfo>
      </xs:annotation>
      <xs:complexType >
         <xs:sequence>
          <xs:element name='identifier'  />
          ...
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name='branchNestedComplex' >
      <xs:annotation>
       <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:discriminator test='{./Header/identifier eq "c"}'/>
        </xs:appinfo>
      </xs:annotation>
      <xs:complexType >
        <xs:sequence>
          <xs:element name='Header'  />
            <xs:complexType >
              <xs:sequence>
                <xs:element name='identifier'  />
                  ...
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:choice>
```

```
</xs:sequence>
```

7.7 DFDL Variable Annotations

DFDL Variables provide a means for communication and parameterization within a DFDL schema. Use of variables increases the modularity of a schema by enabling some parts of a schema to be parameterized so that they are reusable.

There are 3 DFDL annotation elements associated with DFDL variables:

- dfdl:defineVariable - defines a variable and creates a global instance of it.
- dfdl:newVariableInstance - creates a scoped instance of a variable.
- dfdl:setVariable - assigns the value of a variable instance, which can be global or scoped.

Variables are defined at the top-level of a schema and have a specific simple type.

A distinction is made between the variable as defined, and an *instance* of the variable where a value can be stored.

The dfdl:defineVariable annotation defines the name, type, and optionally default value for the variable. It is like defining a class of variables, instances of which actually store values. The dfdl:defineVariable also introduces a single unique global instance of the variable. Additional instances may be allocated in a stack-like fashion using dfdl:newVariableInstance which causes new instances to come into existence upon entry to the scope of a model group, and these instances go away on exit from the same.

DFDL variables only *vary* in the sense that different instances of the same variable can have different values. A single instance of a variable only ever takes on a single value. Each variable instance is a *single-assignment* location for a value[9]. Once a variable instance's value has been read, it can never be assigned again. If it has not yet been assigned, and its default value has not been read, then a variable instance can be assigned *once* using dfdl:setVariable.

Variables are used by referencing them in DFDL expressions by prefixing their QNames with '$'.

More information about variables and how they work operationally is in Section 18.2 Variables. The remaining sub-sections of this section focus only on the variable-related DFDL annotations and their syntax.

7.7.1 dfdl:defineVariable Annotation Element

A global variable is introduced using dfdl:defineVariable:

```
<dfdl:defineVariable
       name = NCName
       type? = QName
      defaultValue? = logical value or dfdl expression
      external? = 'false' | 'true' >
  <!-- Contains: logical value or dfdl expression (default value) -->
</dfdl:defineVariable>
```

The name of a newly defined variable is placed into the target namespace of the schema containing the annotation. Variable names are distinct from format and escape scheme names and so cannot conflict with them. A variable can have any type from the DFDL subset of XML schema simple types. If no type is specified, the type is xs:string.

The defaultValue is optional. This is a literal value or an expression which evaluates to a constant, and it can be specified as an attribute or as the element value. If specified, the default value must match the type of the variable (otherwise it is a Schema Definition Error). If the defaultValue is given by an expression that expression must not contain any relative path (otherwise it is a Schema Definition Error).

Note that the syntax supports both a defaultValue attribute and the default value being specified by the element value. Only one or the other may be present (otherwise it is a Schema Definition Error). To set the default value to "" (empty string), the defaultValue attribute syntax must be used, or the expression { "" } must be used as the element value.

Note also that the value of the name attribute is an NCName (non-colon name - that is, may not have a prefix). The name of a variable is defined in the target namespace of the schema containing the definition. If multiple dfdl:defineVariable definitions have the same 'name' attribute in the same namespace then it is a Schema Definition Error.

[9] The rationale for single-assignment variables is to keep DFDL schemas *declarative* by preventing variables from being used as algorithmic accumulators. See the Appendix B: Rationale for Single-Assignment Variables.

A default *instance* of the variable is automatically created (with global scope) at the start of a DFDL parse or unparse. Additional instances of a variable can be created with the scope of other schema components. See Section 7.7.2 The dfdl:newVariableInstance Statement Annotation Element.

The external property is optional. If not specified it takes the default value 'false'. If true, the value may be provided by the DFDL processor and this external value is used as the global default value overriding any defaultValue specified on the dfdl:defineVariable annotation. The mechanism by which the processor provides this value is *implementation-defined*.

A variable instance gets its value either from the default value provided in the dfdl:defineVariable definition, from an external binding of the variable if the definition has the external attribute, from a dfdl:setVariable statement (See Section 7.7.3, The dfdl:setVariable Statement Annotation Element), or from the default value of a dfdl:newVariableInstance statement (See Section 7.7.2 The dfdl:newVariableInstance Statement Annotation Element.)

There is no required order between dfdl:defineVariable and other schema level defining annotations or a dfdl:format annotation that may refer to the variable.

A defaultValue expression MUST be evaluated before processing of the data stream begins.

A defaultValue expression can refer to other variables but not to the Infoset (so no path locations).When a defaultValue expression references other variables, the referenced variables each must either have a defaultValue or be external. It is a Schema Definition Error otherwise.

If a defaultValue expression references another variable then the single-assignment nature of variables prevents the referenced variable's value from ever changing, that is, it is considered to be a read of the variable's value, and once read, a variable's value cannot be changed.

If a defaultValue expression references another variable and this causes a circular reference, it is a Schema Definition Error.

It is a Schema Definition Error if the type of the variable is a user-defined simple type restriction.

7.7.1.1 Examples

```
 <dfdl:defineVariable name="EDIFACT_DS" type="xs:string"
                      defaultValue="," />

<dfdl:defineVariable name="codepage" type="xs:string"
                      external="true">utf-8</dfdl:defineVariable>
```

7.7.1.2 Predefined Variables

The following variables are predefined, and their names are in the DFDL namespace (http://www.ogf.org/dfdl/dfdl-1.0/)

Name	Type	Default value	External
dfdl:encoding	xs:string	'UTF-8'	true
dfdl:byteOrder	xs:string	'bigEndian'	true
dfdl:binaryFloatRep	xs:string	'ieee'	true
dfdl:outputNewLine	xs:string	'%LF;'	true

Table 9 Pre-defined variables

These variables are expected to be commonly set externally so are predefined for convenience. Below the DFDL encoding property is being set to the value of a DFDL expression (between "{" and "}"), and that expression just returns the value of the dfdl:encoding variable which is being referenced as $dfdl:encoding below.

```
      <xs:element name="title" type="xs:string">
        <xs:annotation>
          <xs:appinfo source="http://www.ogf.org/dfdl/">
            <dfdl:element encoding="{$dfdl:encoding}" />
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
```

7.7.2 The dfdl:newVariableInstance Statement Annotation Element

Scoped instances of defined variables are created using dfdl:newVariableInstance:

```
<dfdl:newVariableInstance
       ref = QName
      defaultValue? = logical value or dfdl expression >
  <!-- Contains: logical value or dfdl expression (value) -->
</dfdl:newVariableInstance>
```

All instances share the same name, type, and default value if provided, but they have distinct storage for separate values using a stack-like mechanism where a new instance is introduced for a model group. These new instances are associated with a schema component using dfdl:newVariableInstance. These instances have the lifetime of the schema component. While that schema component is being parsed/unparsed, the new variable instance is used and other scoped variable instances for the same variable are not available.

Since an initial global instance is created when the variable is defined, the use of dfdl:newVariableInstance is optional.

The dfdl:newVariableInstance annotation can be used on a group reference, sequence or choice only. It is a Schema Definition Error otherwise.

The lifetime of the instance of a variable is the *dynamic scope* of the schema component and its content model and so is inherited by any contained constructs or construct references.

The ref property is a QName. That is, it may be qualified with a namespace prefix.

An optional defaultValue *for the instance* may be specified. It can be specified as an attribute or as the element value. The expression must not contain forward references to elements which have not yet been processed nor to the current component. If specified the default value must match the type of the variable as specified by dfdl:defineVariable. If the instance is not assigned a new default value then it inherits the default value specified by dfdl:defineVariable or externally provided by the DFDL processor. If a default value is not specified (and has not been specified by dfdl:defineVariable) then the value of this instance is undefined until explicitly set (using dfdl:setVariable).

If a default value is specified this initial value of the instance is created when the instance is created. The value overrides any (global) default value which was specified by dfdl:defineVariable or which was provided externally to the DFDL processor. A variable instance with a valid value (specified or default) can be referenced anywhere within the scope of the element on which the instance was created.

Note that the syntax supports both a defaultValue attribute and the default value being specified by the annotation element value. Only one or the other may be present. (Schema definition error otherwise.)

To set the default value to "" (empty string), the defaultValue attribute syntax must be used, or the expression { "" } must be used as the element value.

The resolved set of annotations for a component may contain multiple dfdl:newVariableInstance statements. They must all be for unique variables; it is a Schema Definition Error otherwise. The order of execution is specified in Section 9.5 Evaluation Order for Statement Annotations.

There is no short form syntax for creating variable instances.

7.7.2.1 Examples

```
<dfdl:newVariableInstance ref="EDIFACT_DS" defaultValue=","/>

<dfdl:newVariableInstance ref="lengthUnitBits">
    { if (../ndb/fmtCode eq "bits") then 1 else 8 }
</dfdl:newVariableInstance>
```

7.7.3 The dfdl:setVariable Statement Annotation Element

Variable instances get their values either by default, by external definition, or by subsequent assignment using the dfdl:setVariable statement annotation.

```
<dfdl:setVariable
       ref = QName
       value? = logical value or dfdl expression >
  <!-- Contains: logical value or dfdl expression (value) -->
</dfdl:setVariable>
```

The dfdl:setVariable annotation can be used on a simple type, group reference, sequence or choice. It may be used on an element or element reference only if the element is of simple type. It is a Schema Definition Error if dfdl:setVariable appears on an element of complex type, or an element reference to an element of complex type.

The ref property is a QName. That is, it may be qualified with a namespace prefix.

The syntax supports both a value attribute and the 'value' being specified by the element value. Only one or the other may be present (otherwise it is a Schema Definition Error). To set the value to "" (empty string), the value attribute syntax must be used, or the expression { "" } must be used as the element value.

The value must match the type of the variable as specified by dfdl:defineVariable.

A dfdl:setVariable value expression may refer to the value of this element using a relative path value ".". Use of relative path expressions is recommended wherever possible as this allows the behavior of the parser to be more effectively scoped. However, this practice is not enforced and there may be situations in which use of an absolute path is in fact necessary.

The expression must not contain forward references to elements which have not yet been processed.

In normal processing, the value of an instance can only be set once using dfdl:setVariable. Attempting to set the value of the variable instance for a second time is a Schema Definition Error. In addition, if a reference to the variable's value has already occurred and returned a default or an externally supplied value, then no assignment (even a first one) can occur. An exception to this behavior occurs whenever the DFDL processor backtracks because it is processing multiple branches of a choice or as a result of speculative parsing. In this case the variable state is also rewound. See Section 9 DFDL Processing Introduction.

A dfdl:setVariable overrides any default value specified on either dfdl:defineVariable or dfdl:newVariableInstance, or externally.

The resolved set of annotations for an annotation point may contain multiple dfdl:setVariable statements. They must all be for unique variables (different name and/or namespace) and it is a Schema Definition Error otherwise. The order of execution is specified in Section 9.5 Evaluation Order for Statement Annotations.

There is no short form syntax for variable assignment.

7.7.3.1 **Examples**

```
<xs:element name="ds" type="xs:string">
   <xs:annotation>< xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:setVariable ref="EDI:EDIFACT_DS" value="{.}" />
      <dfdl:setVariable ref="delimiter"> {.} </dfdl:setVariable>
   </xs:appinfo></xs:annotation>
</xs:element>
```

In the above example, the element named "ds" contains the string to be used as the EDI:EDIFACT_DS delimiter at other places in the data, so the above defines the value of the EDI:EDIFACT_DS variable to take on the value of this element. The variable delimiter (in the default namespace) is also being assigned the same value using other syntax.

8 Property Scoping and DFDL Schema Checking

8.1 Property Scoping

8.1.1 Property Scoping Rules

This section describes the rules that govern the scope over which DFDL representation properties apply

The scope of the representational properties on each of the component format annotations is given in **Table 10 DFDL annotation scoping**

Annotation Point	Property Scope
Schema declaration	dfdl:format representation properties apply *lexically* as default properties over all components in the schema
Element declaration	dfdl:element properties apply locally
Element reference	dfdl:element properties apply locally
Simple type definition	dfdl:simpleType properties apply locally
Sequence	dfdl:sequence properties apply locally
Choice	dfdl:choice properties apply locally
Group reference	dfdl:group properties apply locally

Table 10 DFDL annotation scoping

Note: This table lists DFDL annotations on schema components. DFDL annotations can also be placed on other DFDL annotations, such as a dfdl:format within a dfdl:defineFormat, to provide a named reusable format definition. In this case the annotation applies only where the named format is referenced.

DFDL representation properties explicitly defined on annotations, other than a dfdl:format on an xs:schema declaration, apply locally to that component only. The explicitly defined properties are the combination of any defined locally on the annotation and any defined on the dfdl:defineFormat annotation referenced by a local dfdl:ref property. When a property is defined both locally and on the dfdl:defineFormat, the locally defined property takes precedence.

The dfdl:format annotation on the top level xs:schema declaration provides defaults for the DFDL representation properties at every DFDL-annotatable component contained in the schema document. They do not apply to any components in any included or imported schema document (these may have their own defaults).

8.1.2 Providing Defaults for DFDL properties

A dfdl:format annotation on the top level xs:schema declaration may provide defaults for some or all the DFDL representation properties at every annotation point within the schema document. The default properties may be specified in attribute or element form. (Short form is not allowed on the xs:schema element.)

The dfdl:ref property is not a representation property so no default can be set.

The dfdl:escapeSchemeRef property provides a default reference to a dfdl:defineEscapeScheme, the properties of dfdl:escapeScheme are not defaulted individually.

DFDL representation properties defined explicitly on a component apply only to that component and override the default value of that property provided by a default format specified by an xs:schema dfdl:format annotation.

The example below demonstrates the overriding of the encoding property. The value `'ASCII'` is the default value for the `title` element, but then it is overridden by the locally defined `utf-8` value for the encoding property, which takes precedence.

```
<xs:schema>
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:format encoding="ASCII" />
    </xs:appinfo>
  </xs:annotation>
```

```
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string">
          <xs:annotation>
            <xs:appinfo source="http://www.ogf.org/dfdl/">
              <dfdl:element encoding="utf-8" />
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name="pages" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

8.1.3 Combining DFDL Representation Properties from a dfdl:defineFormat

The DFDL representation properties contained in a referenced dfdl:defineFormat are combined with any DFDL representation properties defined locally on a construct as if they had been defined locally. If the same property is defined locally in and in the referenced dfdl:defineFormat then the local property takes precedence. The combined set of explicit DFDL properties has precedence over any defaults set by a dfdl:format on the xs:schema.

```
<xs:schema>
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:defineFormat name='myFormat'>
        <dfdl:format encoding="ASCII" />
      </dfdl:defineFormat>
    </xs:appinfo>
  </xs:annotation>

  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string">
          <xs:annotation>
            <xs:appinfo source="http://www.ogf.org/dfdl/">
              <dfdl:element ref='myFormat' encoding="UTF-8" />
            </xs:appinfo>
          </xs:annotation>
        </xs:element>
        <xs:element name="pages" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The example above demonstrates the overriding of an encoding property. The 'ASCII' format encoding from the 'myFormat' is overridden by the UTF-8 format encoding, which as a locally defined property takes precedence.

8.1.4 Combining DFDL Properties from References

The DFDL properties from the following types of reference are combined using the rules below:

- An xs:element and its referenced xs:simpleType restriction
- An xs:element reference and its referenced global xs:element
- An xs:group reference and an xs:sequence or xs:choice in its referenced global xs:group
- An xs:simpleType restriction and its base xs:simpleType restriction

Rules

1. Create (a) an empty working set of "explicit" properties, and (b) an empty working set of "default" properties.
2. Move to the innermost schema component in the chain of references.
3. Assemble its applicable "explicit" properties from its local dfdl:ref (if present) and its local properties (if present), the latter overriding the former (that is, local wins over referenced).

4. Combine these with the current working set of "explicit" properties. It is a Schema Definition Error if the same property appears twice. The result is a new working set of "explicit" properties.
5. Obtain applicable "default" properties from a dfdl:format annotation on the xs:schema that contains the component (if such annotation is present). Combine these with the current working set of "default" properties, the latter overriding the former (that is, inner wins). Result is a new working set of "default" properties.
6. Move to the schema component that references the current component and repeat starting at step 3. If there is no referencing component, carry out step 5 and then go to step 7.
7. Combine the resultant sets of properties. The "explicit" properties take priority, "defaults" only used when no "explicit" property is present. It is a Schema Definition Error if a required property is in neither the "explicit" nor the "default" working sets.

The "Applicable" properties are all the DFDL properties that apply to that schema component. For example, all the DFDL properties that apply to a particular xs:simpleType (as defined by Section 13).

```
<xs:simpleType name="newType">
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:simpleType alignment="16"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:maxInclusive value="10"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="testElement1" type="newType">
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:element representation="binary"/>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

The locally defined dfdl:alignment property with value '16' from the xs:simpleType 'newType' is combined with the locally defined dfdl:representation property with value 'binary' and applied to element 'testElement1',

```
<xs:simpleType name="otherNewType">
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:simpleType alignment="64"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:restriction base="newType">
    <xs:maxInclusive value="5"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="newType">
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:simpleType representation='binary'/>
    </xs:appinfo>
  </xs:annotation>
  <xs:restriction base="xs:int">
    <xs:maxInclusive value="10"/>
  </xs:restriction>
</xs:simpleType>
```

The locally defined dfdl:representation property with value 'binary' is combined with the locally defined dfdl:alignment property with value '64' from the xs:simpleType restriction 'otherNewType'.

```
<xs:sequence>
  <xs:element ref="testElement1">
    <xs:annotation>
      <xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:element binaryNumberRep ="binary"/>
      </xs:appinfo>
    </xs:annotation>
```

```
  </xs:element>
</xs:sequence>

<xs:element name="testElement1" type="newType">
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:element representation="binary"/>
    </xs:appinfo>
  </xs:annotation>
</xs:element>

<xs:simpleType name="newType">
  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:simpleType alignment="16"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:restriction base="xs:int">
    <xs:maxInclusive value="10"/>
  </xs:restriction>
</xs:simpleType>
```

The locally defined dfdl:alignment property with value '16' from the xs:simpleType 'newType' is combined with the locally defined dfdl:representation property with value 'binary' and locally defined dfdl:binaryNumberRep with a value of 'binary'

```
<!-- SCHEMA1 -->
<xs:schema targetNamespace="" xmlns:tns1="http://tns1">

  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:format encoding="ASCII" byteOrder="littleEndian"
                   initiator="" terminator=""
                   sequenceKind="ordered"  />
    </xs:appinfo>
  </xs:annotation>

  <xsd:import namespace="http://tns2" schemaLocation="SCHEMA2.xsd"/>

  <xs:element name="book">
    <xs:complexType>
      <xs:group ref="tns2:ggrp1" dfdl:separator=","></xs:group>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

```
<!-- SCHEMA2 -->
<xs:schema targetNamespace="" xmlns:tns2="http://tns2">

  <xs:annotation>
    <xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:format encoding="UTF-8" byteOrder="littleEndian"
                   initiator=""
                   sequenceKind="ordered"  />
    </xs:appinfo>
  </xs:annotation>

  <xs:group name="ggrp1" >
    <xs:sequence dfdl:separatorPosition="infix" >
      <xs:element name="customer" type="xs:string"
              dfdl:length="8" dfdl:lengthKind="explicit" />
    </xs:sequence>
  </xs:group>
</xs:schema>
```

The DFDL properties applied to the xs:sequence in xs:group "ggrp1" in SCHEMA2 when referenced from the group reference in SCHEMA1 are

1. dfdl:separator "," from the group reference in SCHEMA1
2. dfdl:separatorPosition "infix" from the group declaration in SCHEMA2
3. dfdl:encoding "UTF-8", dfdl:initiator "" from the default dfdl:format annotation in SCHEMA2
4. dfdl:terminator "" from the default dfdl:format annotation in SCHEMA1

8.2 DFDL Schema Checking

When the DFDL schema itself contains an error, it implies that the DFDL processor cannot process data because the DFDL schema is not meaningful. All conforming DFDL processors MUST detect all Schema Definition Errors and MUST issue appropriate diagnostic messages. The behavior of a DFDL processor after a Schema Definition Error is detected is out of scope for this specification. There is no centralized listing of the Schema Definition Errors; they are defined throughout this specification.

When a Schema Definition Error can be detected *statically*, that is given only the schema, it is desirable, though not required by the DFDL 1.0 specification, that diagnostic messages SHOULD be issued before any data are processed. However, because some representation properties may obtain their values from the data, not all Schema Definition Errors can be detected without reference to data so some Schema Definition Error diagnostics MAY of necessity be issued once data is being processed.

The expression language included within DFDL is strongly, statically type checkable. This means that type checking of expressions MAY be performed statically, that is, without processing data, and implementations are encouraged to perform this checking statically so that *Static Type Errors* (Schema Definition Errors having to do with type inconsistencies) can be detected before processing data.

8.2.1 Schema Component Constraint: Unique Particle Attribution

The term *particle* is used in XSD to refer to a schema component that can have dimension (XSD minOccurs and/or XSD maxOccurs) expressed on it. In DFDL only local element declarations and element references are particles.

A DFDL processor MUST implement the Schema Component Constraint: Unique Particle Attribution defined in *XML Schema Part 1: Structures* [XSDLV1] that applies to the DFDL schema subset.

Two elements **overlap** if

- They are both element declaration particles whose declarations have the same name and target namespace.

A schema violates the unique attribution constraint if it contains two particles which overlap and which either

- are both in the particles of a *choice* group.

or

- either describes adjacent information items in an xs:sequence and the first has XSD minOccurs less than XSD maxOccurs.

8.2.2 Optional Checks and Warnings

- A DFDL processor that only implements a DFDL parser does not have to perform Schema Definition Error checking for properties that are solely used when unparsing, though it is RECOMMENDED that it does so for portability reasons.
- A DFDL processor that does not implement some optional DFDL language features does not have to check properties or annotations needed by those optional language features but MUST issue a warning that an unrecognized property or annotation has been encountered.
- A DFDL processor MUST NOT check global element declarations nor type or group definitions as they may legitimately be incomplete due to properties intended to be supplied based on scoping rules and the context at the point of use. There are two exceptions to this, which MUST be checked:
    1. Global simple type definitions that are referenced by the dfdl:prefixLengthType property
    2. Global element declarations that are the document root.

Some situations suggest likely errors, but a DFDL processor cannot be certain. In these situations, a DFDL processor MAY issue warnings to assist a DFDL schema author in identifying likely errors. An important case of this is when the DFDL processor encounters a schema component and annotation where there are explicitly properties that are not relevant to the component as defined. Depending on the specifics of the component and property the DFDL processor MUST take certain actions. If the:

- Property is not applicable to the component's DFDL annotation.
    - Schema Definition Error. Example is dfdl:lengthKind on xs:sequence.

However, for these situations, the DFDL processor MAY take certain actions:

- Property is not applicable because of simple type.
    - Warning (optional). Example is dfdl:calendarPatternKind on xs:string.
- Property is not applicable because of another DFDL property setting.
    - Warning (optional). Example is dfdl:binaryNumberRep when dfdl:representation is text.
- Invalid value for a property that is unused or ignored.
    - Warning (optional). Example is dfdl:lengthKind is not ‘explicit’ but dfdl:length is an expression and that expression contains invalid syntax.

9 DFDL Processing Introduction

A *DFDL Parser* is an application or code library that takes as input:

- A DFDL annotated XML schema
- A data stream

It uses the DFDL schema description to interpret the data stream and realize the DFDL Information Set. If successful the data stream is said to be *well-formed* for the data format described by the DFDL Schema. The information set can then be written out (for example it could be realized as an XML or JSON text string) or it can be accessed by an application through an API (for example, a DOM-like tree could be created in memory for access by applications).

Symmetrically, there is a notion of a *DFDL Unparser*. The unparser works from an instance of the DFDL Information Set, a DFDL annotated schema and writes out to a target data stream in the appropriate representation formats.

Often both parser and unparser are implemented in the same body of software and so are not always distinguished. Collectively they are called a *DFDL Processor*. The parser and unparser MAY, of course, be different bodies of software. Conforming DFDL processors MAY implement only a parser, because the unparser is an optional feature of DFDL.

9.1 Parser Overview

The DFDL logical parser is a recursive-descent parser[10] having guided, but potentially unbounded look ahead. A DFDL parser reads a specification (the DFDL schema) and it recursively walks down and up the schema as it processes the data. This is done in a manner consistent with the scoping of properties and variables described in Section 8 Property Scoping and DFDL Schema Checking.

Property Scoping

Property Scoping RulesThe unbounded look ahead means that there are situations where the parser MUST speculatively attempt to parse data where the occurrence of a Processing Error causes the parser to suppress the error, back out and make another attempt.

Implementations of DFDL MAY provide control mechanisms for limiting the speculative search behavior of DFDL parsers. The nature of these mechanisms is beyond the scope of the DFDL specification which defines the behavior of conforming parsers only on data that does not cause an implementation to reach such a control-mechanism limit. Any such control mechanisms MUST be documented by the implementation and are thus *implementation-defined*.

The logical parser recursively descends the DFDL schema beginning with the global element that is the document root. This is specified for the processor in an implementation-defined manner, see Section 20 External Control of the DFDL Processor. Depending on the kind of schema construct that is encountered and the DFDL annotations on it, and the pre-existing context, the parser performs specific parsing operations on the data stream. These parsing operations typically recognize and consume data from the stream and construct values in the logical model. For values of complex types and for arrays, these logical model values may incorporate values created by recursive parsing.

DFDL Implementations are free to use whatever techniques for parsing they wish so long as the semantics are equivalent to that of the speculative recursive-descent logical parser described in this specification. Implementations MUST distinguish the various kinds of errors (Schema Definition Error, Processing Error, etc.) no matter what time they are detected. Some implementations MAY not detect certain Schema Definition Errors until data are being parsed; however, they MUST still distinguish Schema Definition Errors from Processing Errors.

9.1.1 Points of Uncertainty

A *point of uncertainty* occurs when there is more than one schema component that might be applied based on parsing up to the current point in the data stream.

Any one of the following constructs is a point of uncertainty:

- An xs:choice

[10] A "top-down" parser built from a set of mutually-recursive procedures or a non-recursive equivalent where each such procedure usually implements one of the productions of the grammar. Thus, the structure of the resulting program closely mirrors that of the grammar it recognizes. See [RDP].

- All xs:elements in an unordered xs:sequence (dfdl:sequenceKind[11] is 'unordered')
- All xs:elements in an xs:sequence containing one or more dfdl:floating[12] xs:elements

Any one of the following constructs is a *potential* point of uncertainty:

- An optional[13] xs:element
- An array xs:element.

Examples of potential points of uncertainty are in Section 9.3.3 Resolving Points of Uncertainty.

9.1.2 Processing Error

If a DFDL schema contains no Schema Definition Errors, then there is the additional possibility of a *Processing Error* when processing data using a DFDL schema. A Processing Error occurs when parsing if the data does not conform to the format described by the schema, that is to say, the data is not well-formed relative to the schema. A Processing Error occurs when unparsing when the incoming Infoset does not conform to the logical structure described by the schema.

Processing Errors interact with the schema's points of uncertainty. When a DFDL parser encounters a Processing Error, then that error is said to be *suppressed* by a point of uncertainty if there is another schema component that can be selected by the parsing algorithm. The details of the DFDL parsing algorithm are described in Section 9.3.

Processing Errors MUST be able to be suppressed by a point of uncertainty. See Section 9.3.3.

Note that unlike Processing Errors, Schema Definition Errors cannot be suppressed by points of uncertainty when parsing data. That is, a Schema Definition Error is fatal. It does not trigger search or backtracking to find alternative ways to parse the data.

9.1.3 Recoverable Error

This error type is used with the dfdl:assert annotation when parsing to permit the checking of physical format constraints without terminating a parse. For example, some formats have redundancy by having known lengths, as well as delimiters. A Recoverable Error can be issued, using an assert to check a physical length constraint when property lengthKind is 'delimited'.

Recoverable Errors are independent of validation, and when resolving points of uncertainty, Recoverable Errors are ignored.

9.2 DFDL Data Syntax Grammar

Data in a format describable via a DFDL schema obeys the grammar given here. A given DFDL schema is read by the DFDL processor to provide specific meaning to the terminals and decisions in this grammar.

The bits of the data are divided into two broad categories:

1. Content
2. Framing

The content is the bits of data that are interpreted to compute a logical value.

Framing is the term used to describe the delimiters, length fields, and other parts of the data stream which are present and may be necessary to determine the length or position of the content of DFDL Infoset items.

Note that sometimes the framing is not strictly necessary for parsing, but adds useful redundancy to the data format, allowing corrupt data to be more robustly detected, and sometimes the framing adds human readability to the data format.

In the grammar tables below, the terminal symbols are shown in bold italic font.

Productions
Document **= SimpleElement | ComplexElement** SimpleElement = SimpleLiteralNilElementRep | SimpleEmptyElementRep | SimpleNormalRep SimpleEnclosedElement = SimpleElement | AbsentElementRep

[11] For dfdl:sequenceKind, see Section 14 Sequence Groups.

[12] For dfdl:floating elements, see Section 14.4 Floating Elements.

[13] For optional and array elements, see Section 16 Properties for Array Elements and Optional Elements.

ComplexElement = ComplexLiteralNilElementRep | ComplexNormalRep |
ComplexEmptyElementRep
ComplexEnclosedElement = ComplexElement | AbsentElementRep

EnclosedElement = SimpleEnclosedElement | ComplexEnclosedElement

AbsentElementRep = ***Absent***

SimpleEmptyElementRep = EmptyElementLeftFraming EmptyElementRightFraming
ComplexEmptyElementRep = EmptyElementLeftFraming EmptyElementRightFraming

EmptyElementLeftFraming = LeadingAlignment ***EmptyElementInitiator*** PrefixLength
EmptyElementRightFraming = ***EmptyElementTerminator*** TrailingAlignment

SimpleLiteralNilElementRep = NilElementLeftFraming [***NilLiteralCharacters*** |
NilElementLiteralContent] NilElementRightFraming
ComplexLiteralNilElementRep = NilElementLeftFraming ***NilLiteralValue***
NilElementRightFraming

NilElementLeftFraming = LeadingAlignment ***NilElementInitiator*** PrefixLength
NilElementRightFraming = ***NilElementTerminator*** TrailingAlignment

NilElementLiteralContent = ***LeftPadding NilLiteralValue*** RightPadOrFill

SimpleNormalRep = LeftFraming PrefixLength SimpleContent RightFraming
ComplexNormalRep = LeftFraming PrefixLength ComplexContent RightFraming

LeftFraming = LeadingAlignment ***Initiator***
RightFraming = ***Terminator*** TrailingAlignment

PrefixLength = SimpleContent | PrefixPrefixLength SimpleContent
PrefixPrefixLength = SimpleContent

SimpleContent = ***LeftPadding*** [SimpleLogicalValue] RightPadOrFill
SimpleLogicalValue = ***SimpleNormalValue*** | ***NilLogicalValue***

ComplexContent = ComplexValue ***ElementUnused***
ComplexValue = Sequence | Choice

Sequence = LeftFraming SequenceContent RightFraming
SequenceContent = [***PrefixSeparator*** EnclosedContent [***Separator*** EnclosedContent]*
PostfixSeparator]

Choice = LeftFraming ChoiceContent RightFraming
ChoiceContent = [EnclosedContent] ***ChoiceUnused***

EnclosedContent = [EnclosedElement | Array | Sequence | Choice] Array = [EnclosedElement [***Separator*** EnclosedElement]* [***Separator*** StopValue]] StopValue = SimpleElement
LeadingAlignment = ***LeadingSkip AlignmentFill*** TrailingAlignment = ***TrailingSkip*** RightPadOrFill = ***RightPadding*** | ***RightFill*** | ***RightPadding RightFill***

Table 11 DFDL Grammar Productions

XML Schema and DFDL properties are used to control constraints on the terminals of the above grammar, as well as repetition (the "*" operator), and alternatives (the "|" operator). For a given set of XML Schema and DFDL properties, and prior data, any terminal may be allowed to be length zero, to contain specific data, or to contain a variety of different admissible data.

Some definitions are needed to cover the range of representations that are possible in the data stream for an occurrence of an element. The representations are:

- Nil Representation
- Empty Representation
- Normal Representation
- Absent Representation

These additional concepts are also defined:

- Zero-Length Representation
- Missing

These definitions are with respect to the grammar above and they do reference some DFDL properties necessary for their definitions. These properties are defined in Sections 11 and beyond.

Some examples follow the definitions.

9.2.1 Nil Representation

An element occurrence has a *nil representation* if the element declaration has XSD nillable property 'true' and the occurrence either:

- conforms to the grammar for SimpleNilLiteralElementRep or ComplexNilLiteralElementRep. Specifically, the ***NilElementInitiator*** and ***NilElementTerminator*** regions must be conformant with property dfdl:nilValueDelimiterPolicy[14]. (If non-conformant it is not a Processing Error and the representation is not nil).
- conforms to the grammar for SimpleNormalRep and its SimpleLogicalValue is ***NilLogicalValue***.

The LeadingAlignment, TrailingAlignment, PrefixLength regions may be present.

9.2.2 Empty Representation

An element occurrence has an *empty representation* if the occurrence does not have a nil representation and it conforms to the grammar for SimpleEmptyElementRep or ComplexEmptyElementRep. Specifically, the ***EmptyElementInitiator*** and ***EmptyElementTerminator*** regions must be conformant with dfdl:emptyValueDelimiterPolicy[15] and the occurrence's SimpleContent or ComplexContent region in the data must be of length zero. (If non-conformant it is not a Processing Error and the representation is not empty).

LeadingAlignment, TrailingAlignment, PrefixLength regions may be present.

The *empty representation* is special in DFDL because when parsing it is used to determine when default values are created in the Infoset. The empty representation can require initiators or terminators be present to enable data formats which explicitly distinguish occurrences with empty string/hexBinary values from occurrences that are *missing* or are *absent*. See Section 9.4 Element Defaults below about default values.

[14] For dfdl:nilValueDelimiterPolicy, see Section 13.16 Properties for Nillable Elements.

[15] For dfdl:emptyValueDelimiterPolicy, see Section 12.2 Properties for Specifying Delimiters.

Hence, the empty representation might not be zero-length. it may require specific non-zero-length syntax in the data stream.

The empty representation is not possible for fixed-length elements with a non-zero length.

9.2.3 Normal Representation

An element occurrence has a normal representation if the occurrence does not have the nil representation or the empty representation and it conforms to the grammar for SimpleNormalRep or ComplexNormalRep.

Note that it is possible for the normal representation to be of zero length, but this can only happen when zero-length is not the nil nor empty representation, and the simple type is xs:string or xs:hexBinary. For all other simple types, the normal representation cannot be zero length.

9.2.4 Absent Representation

Often,it is possible to know the location where an element or group's representation would be in the data based on the delimiters of an enclosing group. (An example: if there are adjacent delimiters of an enclosing sequence.) When this location in the data, which is of zero length, cannot be a nil, empty, or normal representation, then it is said to have *absent representation*, or "the representation is absent".

More formally, an element occurrence has an absent representation if the occurrence does not have a nil or empty or normal representation, and it conforms to the grammar for AbsentElementRep. Specifically, the occurrence's representation in the data stream must be of length zero. Consequently, the Initiator, Terminator, LeadingAlignment, TrailingAlignment, PrefixLength regions must not be present.

As an example of an absent representation: during unparsing, if an optional element does not have an item in the Infoset then nothing is output. However, if a separator of an enclosing structure is subsequently output as the immediate next thing, then a subsequent parse of the element may return a representation of length zero. If this happens, and this zero-length representation does not conform to any of the nil representation, the empty representation, or the normal representation, then it is the absent representation, and it behaves *as if the element occurrence is 'missing'*. (The term 'missing' is defined below.)

9.2.5 Zero-length Representation

The term *zero-length representation* is used to describe the situations where any of the above representations turn out to be of length zero due to specific combinations of data type and format properties:

- The nil representation can be a zero-length representation if dfdl:nilValue is ‘%ES;’ or ‘%WSP*;’ appearing on its own as a literal nil value and there is no framing or framing is suppressed by dfdl:nilValueDelimiterPolicy.
- The empty representation can be a zero-length representation if there is no framing or framing is suppressed by dfdl:emptyValueDelimiterPolicy.
- The normal representation can be a zero-length representation if the type is xs:string or xs:hexBinary and there is no framing.
- The absent representation always has a zero-length representation.

If the nil representation may be zero-length, then the absent representation cannot occur because zero-length is interpreted as nil representation.

If the nil representation may not be zero length, but the empty representation is zero-length, then the absent representation cannot occur because zero-length is interpreted as the empty representation.

If the nil and empty representations cannot be zero-length, but the normal representation may be zero length then the absent representation cannot occur because zero length is interpreted as a normal representation.

If the nil representation may not be zero-length, the empty representation may not be zero-length, and the normal representation may not be zero-length, then a zero-length representation is the absent representation, or "is absent".

9.2.6 Missing

When parsing, an element occurrence is missing if it does not have nil, empty, or normal representations, or it has the absent representation.

When parsing, the term missing really covers two situations. First, it subsumes absent representation. Secondly it applies when an element does not have a representation at all in the data stream, that is, when there are insufficient constructs in the data stream to determine the location of the representation of the element; hence, none of the concepts above apply. This is made clearer in the examples below. If an element occurrence is missing when parsing, no item is ever added to the Infoset.

When unparsing, an element occurrence is missing if there is no item in the Infoset. For a required element occurrence, it is this condition that can trigger the creation of a default value in the augmented Infoset. See

Section 9.4 Element Defaults below about default values. For an optional element occurrence, no item is ever added to the augmented Infoset nor any representation ever output in the data stream.

9.2.7 Examples of Missing and Empty Representation

The following examples illustrate missing and empty representation.

```
<xs:sequence dfdl:separator="," dfdl:terminator="@"
             dfdl:separatorSuppressionPolicy="trailingEmpty" ...>
      <xs:element name="A" type="xs:string"
                  dfdl:lengthKind="delimited"/>
      <xs:element name="B" type="xs:string" minOccurs="0"
                  dfdl:lengthKind="delimited"/>
      <xs:element name="C" type="xs:string" minOccurs="0"
                  dfdl:lengthKind="delimited"/>
</xs:sequence>
```

In data stream '`aaa,@`' element B has the empty representation, and element C does not have a representation so is missing.

```
<xs:sequence dfdl:separator=","
             dfdl:separatorSuppressionPolicy="trailingEmpty"...>
      <xs:element name="A" type="xs:string"
                  dfdl:lengthKind="delimited" dfdl:initiator="A:"
                  dfdl:emptyValueDelimiterPolicy=initiator"/>
      <xs:element name="B" type="xs:string" minOccurs="0"
                  dfdl:lengthKind="delimited" dfdl:initiator="B:"
                  dfdl:emptyValueDelimiterPolicy="initiator"/>
      <xs:element name="C" type="xs:string" minOccurs="0"
                  dfdl:lengthKind="delimited" dfdl:initiator="C:"
                  dfdl:emptyValueDelimiterPolicy=initiator"/>
</xs:sequence>
```

In data stream '`A:aaaa,C:cccc`' element B does not have a representation at all, so is missing.

In data stream '`A:aaaa,B:,C:cccc`' element B has the empty representation. The format definition requires element B to have its initiator in order to indicate the empty representation.

In the data stream '`A:aaaa,,C:cccc`' element B has the absent representation, because the processor is able to tell where element B would appear, but the syntax there does not contain the needed initiator delimiter; hence, it does not satisfy any of nil, empty, or normal representation. Since the processor knows its location, and the data stream there (between the two separators) is zero-length, it is the absent representation, and so is missing.

9.2.8 Round Trip Ambiguities

The overlapping nature of the possible representations: normal, empty, nil, and absent, creates a number of ambiguities where taking an Infoset, unparsing it, and reparsing it results in a second Infoset that is not the same as the original. However, taking the second Infoset, unparsing it, and reparsing it, results in a third Infoset which is the same as the second.

When unparsing, if a string Infoset item happens to contain a string that matches either one of the dfdl:nilValue list values or the default value, it is not given any special treatment. The string's characters are output, or if the value is the empty string, zero length content is output. (In both cases along with an initiator or terminator if applicable.) This creates an ambiguity where one can unparse an Infoset item which has member **[nilled]** true, but when reparsed produces an Infoset item which has member **[nilled]** false.

These ambiguities are natural and unavoidable. For example, if the dfdl:nilValue is the 3-character string "nil", then encountering the characters "nil" in the data stream results in an Infoset item with **[nilled]** true. If a processor unparsed a string Infoset item with contents of the 3 characters "nil", this is output as the letters "nil", which on parse does not produce a string with the characters "nil", but rather an Infoset item with no data value and member **[nilled]** true.

To avoid this issue, one can use validation, along with a pattern that prevents the string from matching any of the nil values.

9.3 Parsing Algorithm

A DFDL parser proceeds by determining the existence of occurrences of schema components. It does this by examining the data and the schema, to:

a) Establish representation
b) Resolve points of uncertainty

These two activities are defined below. They are mutually recursive in the expected way as a DFDL schema is a recursive nest of schema components.

The parsing algorithm described here has many aspects which depend on the definitions of numerous DFDL properties. The properties are defined in sections 10 and beyond.

Establishing the representation of an occurrence of a schema component and resolving points of uncertainty involve the concepts of *known-to-exist* and *known-not-to-exist*.

9.3.1 Known-to-exist and Known-not-to-exist

9.3.1.1 Known-to-exist

An occurrence of a schema component is said to be *known-to-exist* when any of these positive determinations hold:

1. There is a dfdl:discriminator[16] applying to the component and its expression evaluates to true or regular expression pattern matches.
2. The component is a direct child of an xs:sequence or xs:choice with dfdl:initiatedContent[17] 'yes' and a dfdl:initiator defined for the component is found.
3. The component is a direct child of an xs:choice with dfdl:choiceDispatchKey[18] and the result of the dfdl:choiceDispatchKey expression matches one of the dfdl:choiceBranchKey property values of the child.

If none of those hold because they are not applicable then the occurrence is still known-to-exist if ALL of the following hold, and no Processing Error occurs during their determination:

1. When there are dfdl:assert[19] statements with failureType 'processingError' on the component, all their expressions evaluate to true or their regular expression patterns match.
2. It has nil, empty, or normal representation.
3. When it has normal representation the content of the representation is convertible to the element type without error.

Note that Validation Errors or Recoverable Errors do not prevent determination that a component is known-to-exist.

9.3.1.2 Processing Error After Determining Known-to-exist

Note that it is possible for an occurrence of a schema component to be known-to-exist due to a positive discrimination, but then subsequently a Processing Error occurs when evaluating a statement annotation such as a dfdl:assert or a dfdl:setVariable, or a Processing Error occurs when determining the representation, or in the case of normal representation and simple type, when converting that representation's content into a value of the type. This Processing Error does not change the fact that the schema component was determined to be known-to-exist. This is important in the discussion in Section 9.3.3, Resolving Points of Uncertainty below.

9.3.1.3 Known-not-to-exist

An occurrence of a schema component is *known-not-to-exist* when any of these negative determinations holds:

1. There is a dfdl:discriminator applying to the component and its expression evaluates to false or regular expression pattern fails to match, or a Processing Error occurs while processing the dfdl:discriminator.
2. The component is a direct child of an xs:sequence or xs:choice with dfdl:initiatedContent 'yes' and an initiator defined for the component is not found.
3. The component is a direct child of an xs:choice with dfdl:choiceDispatchKey and the result of the dfdl:choiceDispatchKey expression does not match any of the dfdl:choiceBranchKey property values of the child.
4. The component is an element of complex type, the model group of which is a sequence group, and the sequence group is known not to exist.

[16] DFDL discriminators are described in Section: 7.6 The dfdl:discriminator Statement Annotation Element.

[17] For dfdl:initiator and dfdl:initiatedContent, see Section 12.2 Properties for Specifying Delimiters.

[18] For dfdl:choiceDispatchKey and dfdl:choiceBranchKey, see Section 15.1.2 Resolving Choices via Direct Dispatch.

[19] DFDL asserts are described in Section 7.5 The dfdl:assert Statement Annotation Element.

If none of those hold because they are not applicable, then a schema component is known-not-to-exist when any of the following hold:

1. The occurrence is missing
2. There is a dfdl:assert with failureType 'processingError' on the component and its expression evaluates to false or its regular expression pattern fails to match, or a Processing Error occurs while processing the dfdl:assert.
3. A Processing Error occurs when parsing the component. Processing Errors include, but are not limited to, inability to identify any of nil, empty, normal or absent representations, or failure to convert a value to the built-in logical type.

Note that Validation Errors or Recoverable Errors do not cause a component to be known-not-to-exist.

Note: based on the above, when processing a sequence for which a separator is defined, the presence of a match in the data for the separator is not sufficient to cause the parser to determine that an associated component is known-to-exist. See Section 14.2 Sequence Groups with Separators for details.

9.3.2 Establishing Representation

Unless an element occurrence is known-not-to-exist, the parsing algorithm establishes if it has the nil, empty, normal, or absent representation.

The first step is to see if the SimpleContent or ComplexContent region is of length zero as a first approximation. This is dfdl:lengthKind dependent.

- explicit => length is zero (either fixed or from expression evaluation)
- prefixed => length given by the prefix is zero
- implicit (simple) => length is zero[20]
- implicit (complex) => not possible.
- delimited => length is zero (in scope delimiter is immediately encountered)
- pattern => pattern returns zero length match
- endOfParent => already positioned at parent's end so length is zero

9.3.2.1 Simple element

If the result is length zero as described above, the representation is then established by checking, in order, for:

1. nil representation (if %ES; or %WSP*; on its own is a literal nil value).
2. empty representation.
3. normal representation (xs:string or xs:hexBinary only)
4. absent representation (if none of the prior representations apply).

If the result is not length zero, the representation is then established by checking, in order, for:

1. nil representation (as a literal nil value)
2. nil representation (as a logical nil value)
3. normal representation

9.3.2.2 Complex element

If the result is length zero as described above, the representation is then established by checking for:

- nil representation (if %ES; is a literal nil value).[21]

To establish any other representations requires that the parser descends into the complex type for the element, and returns successfully (that is, no unsuppressed Processing Error occurs). If the result is zero bits consumed, the representation is then established by checking, in order, for:

1. empty representation.
2. absent representation (if none of the prior representations apply).

[20] This is a corner case that only happens when type is xs:string or xs:hexBinary and the maxLength facet is 0. Such an element can only be of length 0.

[21] It is a Schema Definition Error if a complex element has XSD nillable ‘true’ and dfdl:lengthKind ‘implicit’.

Otherwise the element has normal representation.

Note: The DFDL parser SHALL NOT recursively parse the schema components inside a complex element when it has already established that the element occurrence is missing[22].

9.3.3 Resolving Points of Uncertainty

A point of uncertainty occurs when there is more than one schema component that might be applied at the current point in the data stream. Points of uncertainty can be nested.

The parser resolves these points of uncertainty by way of a set of construct-specific rules given below along with determining whether schema components are known-to-exist or known-not-to-exist. For some of these constructs, whether there is an actual point of uncertainty depends on the representation of the constructs in the data.

An xs:choice is always a point of uncertainty. It is resolved sequentially, or by direct dispatch. Sequential choice resolution occurs by parsing each choice branch in schema definition order until one is known-to-exist. It is a Processing Error if none of the choice branches are known-to-exist. Direct-dispatch choice resolution occurs by matching the value of the dfdl:choiceDispatchKey property to the value of one of the dfdl:choiceBranchKey property values of one of the choice branches. It is a Processing Error if none of the choice branches have a matching value in their dfdl:choiceBranchKey property.

An element in an unordered xs:sequence is always a point of uncertainty. It is resolved by parsing for the child components of the sequence in schema definition order at each point in the data stream where a component can exist until the required number of occurrences of each child component is known-to-exist or the sequence is terminated by delimiters or specified length.

An element in a sequence with one or more floating elements is always a point of uncertainty. It is resolved by parsing for the expected element at that point in the data stream. If the expected element is known-not-to-exist then an occurrence of each floating element is parsed in schema definition order.

When parsing an array or optional element, points of uncertainty only occur for certain values of dfdl:occursCountKind[23], as follows:

dfdl:occursCountKind	Details of Point of Uncertainty
fixed	No point of uncertainty (XSD maxOccurs occurrences expected).
implicit	A point of uncertainty exists after XSD minOccurs occurrences are found and until XSD maxOccurs occurrences are found.
parsed	A point of uncertainty exists for all occurrences
expression	No point of uncertainty (dfdl:occursCount[24] values are expected)
stopValue	No point of uncertainty (The stop value must always be present, even when XSD minOccurs is 0).

Table 12: Points of Uncertainty and dfdl:occursCountKind

An optional element point of uncertainty is resolved by parsing the element until it is either known-to-exist or known-not-to-exist. Whether an optional element is an actual point of uncertainty depends on property dfdl:occursCountKind as described above.

For an array element, the point of uncertainty is resolved for each occurrence separately by parsing the occurrence until it is either known-to-exist or known-not-to-exist.

9.3.3.1 Nested Points of Uncertainty

A point of uncertainty can be resolved because a schema component has been determined to be known-to-exist due to positive discrimination. In that case, if a subsequent Processing Error occurs when completing the parsing of that schema component this causes the next enclosing schema component surrounding this point of uncertainty to be determined to be known-not-to exist.

For example, when parsing an element occurrence for an array with a variable number of occurrences, a positive discrimination tells the parser that the currently-being-parsed occurrence is known-to-exist. If a

[22] The rationale for this is that otherwise this could give rise to misleading error messages where the parser reported that required child elements were missing required occurrences. (This is consistent with XML Schema validation, where if a required element is missing, it gets reported as such, and there is nothing reported about its children).

[23] Property dfdl:occursCountKind is defined in Section 16.1.

[24] Property dfdl:occursCount is defined in Section 16.

subsequent Processing Error occurs while completing the parsing of this occurrence, then the entire array is then known-not-to-exist.

Another example is a choice. If a discriminator resolves the choice point of uncertainty to the first of the choice's alternatives, a subsequent Processing Error causes the entire choice construct to be determined to be known-not-to-exist.

This causes the next enclosing point of uncertainty to try the next possible alternative, or if there isn't one, causes an unsuppressed Processing Error.

The behavior of a DFDL processor on an unsuppressed Processing Error is not specified, but it is allowable for implementations to abort further parsing. Any other behavior is implementation-defined.

A discriminator always resolves the nearest enclosing point of uncertainty that is unresolved. If more than one discriminator is evaluated, the first resolves the nearest enclosing point of uncertainty, the second the next nearest enclosing point of uncertainty, and so on.

9.4 Element Defaults

A DFDL processor can create element defaults in the Infoset for both simple and complex elements. This happens quite differently for parsing and unparsing as is explained in this section.

9.4.1 Definitions

9.4.1.1 Default Value

A simple element has a default value if any of these are true:

1. The XSD default property exists. The default value is the XSD default property's value.
2. The XSD fixed[25] property exists. The default value is the XSD fixed property's value.
3. The element has XSD nillable is 'true' and dfdl:useNilForDefault[26] is 'yes'. The corresponding Infoset item has the **[nilled]** member true, and the **[dataValue]** member has no value.

9.4.1.2 Required/Optional Occurrence

An occurrence of an element with an index less than or equal to XSD minOccurs is said to be a *required occurrence*.

An occurrence of an element with an index greater than XSD minOccurs is said to be an *optional occurrence*.

9.4.2 Element Defaults When Parsing

If *empty* representation is established when parsing, the possibility of applying an element default arises. Essentially, if a required occurrence of an element has empty representation, then an element default is applied if present, though there are a couple of variations on this rule. Remember that in order to have established empty representation, the occurrence must be compliant with the dfdl:emptyValueDelimiterPolicy for the element, and for a complex element the parser must have descended into the type and returned with no unsuppressed Processing Error.

The rules for applying element defaults are not dependent on dfdl:occursCountKind. However, if a required occurrence does not produce an item in the Infoset after the rules have been applied, then whether it is a Processing Error or a Validation Error (if validation is enabled) *does* depend on dfdl:occursCountKind (see Section 16.1 dfdl:occursCountKind property).

The sections below indicate when an item is added to the Infoset, and whether it has a default or other value. If there is no Processing Error then regardless of whether an item is added to the Infoset or not, any side-effects due to dfdl:discriminator statements evaluating to true, or dfdl:setVariable statements, are retained.

Assuming the empty representation has been established, there are three cases to consider:

- Simple element (not type xs:string or xs:hexBinary)
- Simple element (type xs:string or xs:hexBinary)
- Complex element

Each is described in a section below.

[25] For the XSD fixed property see Section 5.3.7.

[26] For dfdl:useNilForDefault see Section 13.16.

9.4.2.1 Simple element (not xs:string and not xs:hexBinary)

Required occurrence: If the element has a default value then an item is added to the Infoset using the default value, otherwise nothing is added to the Infoset.

Optional occurrence: Nothing is added to the Infoset.

9.4.2.2 Simple element (xs:string or xs:hexBinary)

Required occurrence: If the element has a default value then an item is added to the Infoset using the default value, otherwise an item is added to the Infoset using empty string (type xs:string) or empty hexBinary (type xs:hexBinary) as the value.

Optional occurrence: if dfdl:emptyValueDelimiterPolicy is applicable and is not 'none'[27], then an item is added to the Infoset using empty string (type xs:string) or empty hexBinary (type xs:hexBinary) as the value, otherwise nothing is added to the Infoset.

Note: To prevent unwanted empty strings or empty hexBinary values from being added to the Infoset, use XSD minLength > '0' and a dfdl:assert that uses the dfdl:checkConstraints()[28] function, to raise a Processing Error.

9.4.2.3 Complex element

Required occurrence: An item is added to the Infoset.

Optional occurrence: if dfdl:emptyValueDelimiterPolicy is applicable and is not 'none'[29], then an item is added to the Infoset, otherwise nothing is added to the Infoset.

For both required and optional occurrences, the parser, by recursive descent, may create the Infoset item and a single child Infoset item. This can occur when:

1. the first child element of the complex type is a required simple element, then an empty string (type xs:string), empty hexBinary (type xs:hexBinary), or default value is alsoadded to the Infoset.
2. the first child element of the complex type is a required complex element, then an item is added to the Infoset (which may itself have a child via (1))

9.4.2.4 Example: Complex Optional Empty Element Not Added to Infoset

Below is an example where an optional complex element with empty representation has nothing added to the infoset. consider the following:

```
<xs:sequence dfdl:separator="|"> <!-- sequence S0 -->
  ...prior schema components ...
  <xs:element name="E1" minOccurs="0"
    dfdl:lengthKind="delimited"
    dfdl:occursCountKind="implicit">
    <xs:complexType>
      <xs:sequence dfdl:separator=";"> <!-- sequence S1 -->
        <xs:element name="E2" type="xs:string" dfdl:lengthKind="delimited"/>
        ... other optional content ...
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:sequence>
```

In the above there is a sequence S0 with a separator that contains among other content an optional, non-nillable, non-initiated, non-terminated element E1 of complex type. The content of the E1 type is a sequence S1 with a different separator and the first child is a required, non-initiated, non-terminated element E2 of type xs:string. The dfdl:lengthKind of both E1 and E2 is 'delimited'.

Now consider a data stream '`...||...`' which has two adjacent S0 separators, and where the parser has successfully parsed the schema components prior to E1 within S0, which is what the "..." prior to the two separators represents. That prior parse is delimited by the first S0 "|" separator, and E1's representation begins immediately after that first S0 separator.

[27] If other than 'none', either an initiator, terminator or both must have been found in the data stream.

[28] For dfdl:checkConstraints function see Section 18.5.3

[29] If other than 'none', either an initiator, terminator or both must have been found in the data stream.

The representation of E1 has zero length because of these two adjacent S0 separators. On processing E1, the parser establishes a point of uncertainty with the data stream positioned after the first S0 separator. The parserthen descends into E1's complex type to process E2. It scans for in-scope delimiters and immediately encounters the second S0 separator. E2 has the empty representation, so E1 is added to the Infoset along with a value of empty string for E2. All other content of S1 is missing, so the parser returns from the descent into E1 with this temporary Infoset (illustrated as XML):

```
<E1>
  <E2></E2>
</E1>
```

Upon this successful parse of E1, it is therefore known-to-exist. However, because the position in the data has not changed, E1 therefore has the empty representation. Because E1 is empty and optional (it has XSD minOccurs='0') and dfdl:emptyValueDelimiterPolicy does not apply, it is not added to the Infoset, and the temporary Infoset item for E1 containing E2 is discarded.

9.4.2.5 Example: Complex Optional Empty Element with Delimiters

This example is similar, but the E1 element has a few additional DFDL properties highlighted in bold below:

```
<xs:sequence dfdl:separator="|"> <!-- sequence S0 -->
  ...prior schema components ...
  <xs:element name="E1" minOccurs="0"
    dfdl:initiator="("
    dfdl:terminator=")"
    dfdl:emptyValueDelimiterPolicy="both"
    dfdl:lengthKind="delimited"
    dfdl:occursCountKind="implicit">
    <xs:complexType>
      <xs:sequence dfdl:separator=";"> <!-- sequence S1 -->
        <xs:element name="E2" type="xs:string" dfdl:lengthKind="delimited"/>
        ... other optional content ...
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:sequence>
```

This changes the definition of element E1 to have an empty representation only if the initiator and terminator are present in the data stream.

Consider has the same data stream '...||...' where there are two adjacent S0 separators. In this case the representation of E1 does not match the empty representation, because the initiator and terminator are not present as the dfdl:emptyValueDelimiterPolicy requires. It also does not have the normal representation, again as the initiator and terminator are not present. E1's representation is absent. Hence, nothing is added to the infoset.

However, if the data stream '...|()|...' is encountered, there are two S0 separators, but between them there are the initiator and terminator of element E1. This satisfies the requirements for the empty representation, but it is not zero length. The recursive parse of E1's complex type constructs these elements (illustrated as XML):

```
<E1>
  <E2></E2>
</E1>
```

These elements for E1 with E2 child would be added to the infoset.

9.4.3 Element Defaults When Unparsing

If an element is *missing* from the Infoset when unparsing, the possibility of applying an element default arises. Essentially if a required occurrence of an element is missing, then an element default is applied if present, and the resulting item is added to the *augmented Infoset* (See Section 9.7*)*

The rules for applying element defaults are not dependent on dfdl:occursCountKind. However if a required occurrence does not produce an item in the augmented Infoset after the rules have been applied then whether it is a Processing Error or a Validation Error (if enabled) *is* dependent on dfdl:occursCountKind (see Section 16.1 dfdl:occursCountKind property).

There are two cases to consider.

9.4.3.1 Simple element

Required occurrence: If an element has a default value then an item is added to the augmented Infoset using the default value, otherwise nothing is added.

Optional occurrence: Nothing is added to the augmented Infoset.

9.4.3.2 Complex element

Required occurrence: An item is added to the augmented Infoset as specified below.

Optional occurrence: Nothing is added to the augmented Infoset.

For a required occurrence, the unparser descends into the complex type:

For a sequence, each child element is examined in schema order and the rules for simple and complex elements applied (recursively). The lack of a default may give rise to a Processing Error, as described above.

For a choice, each branch is examined in schema order and the above rules applied recursively to the branch. The lack of a default may give rise to a Processing Error, as described above, and if so the error is suppressed and the next branch is tried, otherwise that branch is selected. It is a Processing Error if no choice branch is ultimately selected. If no choice branch is selected, then there must be a choice branch with no required elements, and the first such branch would be selected.

9.5 Evaluation Order for Statement Annotations

Given a component of a DFDL schema, there is a resolved set of annotations for it.

Of these, some are statement annotations and the order of their evaluation relative to the actual processing of the schema component itself (parsing or unparsing via its format annotations) is as defined in the ordered lists below.

For elements and element references:

1. dfdl:discriminator or dfdl:assert(s) with testKind 'pattern' (parsing only)
2. dfdl:element following property scoping rules, which includes establishing representation as described in Section 9.3.2 and conversion to the element type for simple types
3. dfdl:setVariable(s) - in lexical order, innermost schema component first
4. dfdl:discriminator or dfdl:assert(s) with testKind 'expression' (parsing only)

For sequences, choices and group references:

1. dfdl:discriminator or dfdl:assert(s) with testKind 'pattern' (parsing only)
2. dfdl:newVariableInstance(s) - in lexical order, innermost schema component first
3. dfdl:setVariable(s) - in lexical order, innermost schema component first
4. dfdl:sequence or dfdl:choice or dfdl:group following property scoping rules and evaluating any property expressions (corresponds to ComplexContent grammar region)
5. dfdl:discriminator or dfdl:assert(s) with testKind 'expression' (parsing only)

The dfdl:setVariable annotations at any one annotation point of the schema are always executed in lexical order. However, dfdl:setVariable annotations can also be found in different annotation points that are combined into the resolved set of annotations for one schema component. In this case, the order of execution of the dfdl:setVariable statements from any one annotation point remains lexical. The order of execution of the dfdl:setVariable annotations different annotation points follows the principle of innermost first, meaning that a schema component that references another schema component has its dfdl:setVariable statements executed *after* those of the referenced schema component. For example, if an element reference and an element declaration both have dfdl:setVariable statements, then those on the element declaration execute before those on the element reference. Similarly, dfdl:setVariable statements on a base simple type execute before those of a simple type derived from it. The dfdl:setVariable statements on a simple type execute before those on an element having that simple type (whether that type is by reference, or when the simple type is lexically nested within the element declaration). The dfdl:setVariable statements on the sequence or choice within a global group definition execute before those on a group reference.

The dfdl:newVariableInstance annotations at any one annotation point of the schema are always executed in lexical order. However, dfdl:newVariableInstance annotations can also be found in different annotation points that are combined into the resolved set of annotations for one schema component. In this case, the order of execution of the dfdl:newVariableInstance statements from any one annotation point remains lexical. The order of execution of the dfdl:newVariableInstance annotations different annotation points follows the principle of innermost first, meaning that a schema component that contains or references another schema component has its dfdl:newVariableInstance statements executed *after* those of the contained or referenced schema component. For example, if a group reference and the sequence or choice group of a group definition both

have dfdl:newVariableInstance statements, then those on the global group definition execute before those on the group reference.

9.5.1 Asserts and Discriminators with testKind 'expression'

Implementations are free to optimize by recognizing and executing discriminators or asserts with testKind 'expression' earlier so long as the resulting behavior is consistent with what results from the description above.

9.5.2 Discriminators with testKind 'expression'

When parsing, an attempt to evaluate a discriminator MUST be made even if preceding statements or the parse of the schema component ended in a Processing Error.

This is because a discriminator's expression can evaluate to true thereby resolving a point of uncertainty even if the complete parsing of the construct ultimately caused a Processing Error.

Such discriminator evaluation has access to the DFDL Infoset of the attempted parse as it existed immediately before detecting the parse failure. Attempts to reference parts of the DFDL Infoset that do not exist are Processing Errors.

9.5.3 Elements and setVariable

The resolved set of dfdl:setVariable statements for an element are executed **after** the parsing of the element. This contrasts with the resolved set of dfdl:setVariable statements for a group which are executed **before** the parsing of the group. (Note that dfdl:setVariable for an element is only allowed on elements of simple type per Section 7.7.3.)

For elements, this implies that these variables are set after the evaluation of expressions corresponding to any computed DFDL properties for that element, and so the variables may not be referenced from expressions that compute these DFDL properties.

That is, if an expression is used to provide the value of a property (such as dfdl:terminator or dfdl:byteOrder), the evaluation of that property expression occurs before any dfdl:setVariable annotation from the resolved set of annotations for that element are executed; hence, the expression providing the value of the property may not reference the variable. Schema authors can insert sequences to provide more precise control over when variables are set.

9.5.4 Controlling the Order of Statement Evaluation

Schema authors can insert xs:sequence constructs to control the timing of evaluation of statements more precisely. For example:

```
<xs:sequence dfdl:separator=",">
   ...
   <xs:element ref="a" .../>
   <xs:sequence>
     <xs:sequence>
       <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/" >
         <dfdl:assert  test="{test expression}" />
       </xs:appinfo></xs:annotation>
     </xs:sequence>
     <xs:element ref="b" .../>
   </xs:sequence>
   ...
</xs:sequence>
```

In the above, the assert test expression is evaluated after parsing element 'a', and before parsing element "b". The use of two nested interior sequences surrounding element 'b' in this manner ensures that the outermost sequence's separator usage is not disrupted.

9.6 Validation

Logical validation checks are constraints expressed in XSD, and they apply to the logical values of the Infoset. Hence, parsing MUST successfully construct the Infoset before validation checks can be performed. This implies that DFDL Validation Errors cannot affect the parsing of data.

DFDL processors MAY provide both validating and non-validating behaviors on either or both of parse and unparse. (A DFDL implementation could support validate on parse, but not support it on unparse and still be considered conforming.)

Validation on unparsing takes place on the augmented Infoset that is created by the unparser as a side-effect of creating the output data stream. Validation errors do not affect unparser behavior.

When resolving points of uncertainty (during parsing), Validation Errors are ignored.

The way a Validation Error is presented to the execution context of a DFDL processor is not specified by the DFDL specification. The validity of an element is recorded in the DFDL Infoset, see Section 4 The DFDL Information Set (Infoset).

The following DFDL schema constructs are allowed in DFDL and are checked if applicable when validating:

1. XSD pattern facet
2. XSD minLength, maxLength
3. XSD minInclusive, minExclusive, maxInclusive, maxExclusive
4. XSD enumeration
5. XSD maxOccurs

Note that validation is distinct from the checking of DFDL assert or discriminator predicates. Both DFDL asserts and discriminators are essential to parsing and are evaluated irrespective of whether validation is enabled or disabled.

There is also a function dfdl:checkConstraints available in the DFDL Expression language. This can be used to explicitly include checking of the XSD constructs as part of parsing a specific element. Such checking is part of parsing and does not create Validation Errors. See Section 18.5.3 DFDL Functions for details.

9.7 Unparser Infoset Augmentation Algorithm

As unparsing progresses and fills in these defaultable and calculated elements, these new item values augment the Infoset, that is, make it bigger.

The unparsing algorithm fills in default values for required elements that are not present, and computes calculated elements by use of the dfdl:outputValueCalc property (see Section 17 Calculated Value Properties).

When unparsing, an element declaration and the Infoset are considered as follows. An implementation MAY use any technique consistent with this algorithm:

a) If the element declaration has a dfdl:outputValueCalc property, then the expression which is the dfdl:outputValueCalc property value is evaluated, and the resulting value becomes the value of the element item in the augmented Infoset. Any pre-existing value for the Infoset item is superseded by this new value.

   References to other augmented Infoset items from within the dfdl:outputValueCalc expression MUST obtain their values from the augmented Infoset directly (when the value is already present) or by recursively using these methods (a) and (b) as needed.

b) If the element declaration has no corresponding value in the augmented Infoset, and the element declaration is for a *required* occurrence, and it *has a default value specified*, then an element item having the default value is created in the augmented Infoset.

c) If any Infoset item's value is requested recursively as a part of (a) above and (a) does not apply, and the corresponding value is not present, and (b) does not apply then it is a Processing Error.

Given this augmented Infoset, then if the element declaration has a corresponding Infoset item then that item is converted to its representation according to its DFDL properties. If the element declaration is for a required occurrence, and there is no value in the augmented Infoset then it is a Processing Error.

10 Overview: Representation Properties and their Format Semantics

The next sections specify the set of DFDL v1.0 properties that may be used in DFDL annotations in DFDL Schemas to describe data formats.

It is a Schema Definition Error when a DFDL schema does *not* contain a definition for a representation property that is needed to interpret the data. For example, a DFDL schema containing any textual data must provide a definition of the character set encoding property (dfdl:encoding) for that textual data, and if it is not part of the format properties context for that data, then it is a Schema Definition Error.

Furthermore, no default values are provided for representation properties as built-in definitions by any DFDL processor. This requires DFDL schemas to be explicit about the representation properties of the data they describe and avoids any possibility of DFDL schemas that are meaningful for some DFDL processors but not others.

The properties are organized as follows:

- Common to both Content and Framing (see 11)
- Common Framing, Position, and Length (see 12)
- Simple Type Content (see 13)
- Sequence Groups (see 14)
- Choice Groups (see 15)
- Array elements and optional elements (see 16)
- Calculated Values (see 17)

Where properties are specific to a physical representation, the property name may choose to reflect this. Where properties are related to a specific logical type grouping (defined below), the property name may choose to reflect this.

A limited number of properties can take a DFDL expression which must return a value of the proper type for the property. Those properties that take an expression explicitly state in the description. Other properties do not take an expression.

The property description defines which schema component that the property may be specified on. In addition, most DFDL properties may be specified on a dfdl:format annotation.

11 Properties Common to both Content and Framing

Property Name	Description
byteOrder	Enum or DFDL Expression Valid values 'bigEndian', 'littleEndian'. This property can be computed by way of an expression which returns the string 'bigEndian' or 'littleEndian'. The expression must not contain forward references to elements which have not yet been processed. Note that there is, intentionally, no such thing as 'native' endian[30]. This property applies to all Number, Calendar (date and time), and Boolean types with representation binary. Specifically, that is binary integers, binary booleans, all packed decimals, binary floats, binary seconds and binary milliseconds. This property is never used to establish the byte order for text /strings, as each character set encoding involving multiple bytes of data per code unit specifies its byte order. Annotation: dfdl:element, dfdl:simpleType
bitOrder	Enum Valid values 'mostSignificantBitFirst', 'leastSignificantBitFirst'. The bits of a byte each have a place value or significance of 2^n, for *n* from 0 to 7. Hence, the byte value 255 = $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$. A bit can always be unambiguously identified as the 2^n-bit. The bit order is the correspondence of a bit's numeric significance to the bit position (1 to 8) within the byte. Value 'mostSignificantBitFirst' means: • The 2^7 bit is first, i.e., has bit position 1. • In general, the 2^n bit has position 8 - *n*. • The least significant bits of byte N are considered to be adjacent to the most significant bits of byte N+1. Value 'leastSignificantBitFirst' means: • The 2^0 bit is first, i.e., has bit position 1. • In general, the 2^n bit has position *n* + 1. • The most significant bits of byte N are considered to be adjacent to the least significant bits of byte N+1. This property applies to all content and framing since it determines which bits of a byte occupy what bit positions. Content and framing are defined in terms of regions of the data stream, and these regions are defined in terms of the starting bit position and ending bit position; hence, dfdl:bitOrder is relevant to determining the specific bits of any grammar region (see Section 9.2 DFDL Data Syntax Grammar) when the region's starting bit position or ending bit position are not on a byte boundary. The bit order can only change on byte boundaries, and alignment of up to 7 bits is skipped (parsing) or inserted (unparsing) to ensure byte-alignment whenever the bit order changes. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
encoding	Enum or DFDL Expression Values are one of:

[30] The concept of native-endian is avoided in DFDL since a DFDL schema containing such a property binding would not contain a complete description of data, but rather an incomplete one which would behave differently based on characteristics of the machine and implementation where the DFDL processor is executed. In DFDL this same behavior is achieved through the use of explicit parameterization using DFDL variables to set dfdl:byteOrder. See Section 7.7.1.2 Predefined Variables.

	• IANA charset name[31] • CCSID[32] • DFDL standard encoding name • Implementation-specific encoding name This property can be computed by way of an expression which returns an appropriate string value. The expression must not contain forward references to elements which have not yet been processed. Note that there is, deliberately, no concept of 'native' encoding[33]. Conforming DFDL v1.0 processors MUST accept at least 'UTF-8', 'UTF-16', 'UTF-16BE', 'UTF-16LE', 'ASCII', and 'ISO-8859-1' as encoding names. The encoding name "UTF-16" is equivalent to "UTF-16BE" and for processors that implement UTF-32, the encoding name "UTF-32" is equivalent to "UTF-32BE". Unlike most other properties with Enum values, encoding names are case-insensitive, so for example 'utf-8', 'Utf-8', and 'UTF-8' are equivalent. The encoding name 'UTF-8' is interpreted strictly and does not include variants such as CESU-8. DFDL standard encoding names are defined in Section 33 Appendix D: DFDL Standard Encodings. When supported, a conforming DFDL implementation MUST implement them in a uniform manner so that they are portable across all DFDL implementations that implement them. Additional implementation-defined encoding names MAY be provided only for character set encodings for which there is no IANA name standard nor CCSID standard nor DFDL standard encoding. These implementation-defined encodings MUST have "X-" as a prefix to their name, as they are subject to being superseded by IANA or DFDL standard encoding names. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
utf16Width	Enum Valid values are 'fixed', 'variable'. Applies only when encoding is 'UTF-16', 'UTF-16BE', UTF16-LE' or their CCSID equivalents. Specifies whether the encoding 'UTF-16' is treated as a fixed or variable width encoding. 'UTF-16' can contain characters which require two codepoints (called a surrogate pair) to represent. When utf16Width is 'fixed', these surrogate code points are treated as separate characters. When utf16Width is 'variable', then surrogate pairs are converted into a single character on parsing, and such a character is split into two characters on unparsing. When utf16Width is 'variable', then on parsing an un-paired surrogate codepoint causes a decode error, which can be controlled via dfdl:encodingErrorPolicy described below. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
ignoreCase	Enum Valid values are 'yes', 'no'. Whether mixed case data is accepted when matching delimiters and data values on input. This affects the behavior of matching for these properties: dfdl:initiator, dfdl:terminator, dfdl:separator, dfdl:nilValue, dfdl:textStandardExponentRep,

[31] IANA is the Internet Assigned Names Authority. See [IANA]

[32] CCSID stands for Coded Character Set ID, a decimal number syntax for a coded character set specifier. [CCSID]

[33] The concept of native character encoding is avoided in DFDL since a DFDL schema containing such a property binding would not contain a complete description of data, but rather an incomplete one which would behave differently based on characteristics of the operating environment where the DFDL processor executes. In DFDL this same behavior is achieved through the use of explicit parameterization using DFDL variables to set dfdl:encoding. See Section 7.7.1.2 Predefined Variables.

	dfdl:textStandardInfinityRep, dfdl:textStandardNaNRep, dfdl:textStandardZeroRep, dfdl:textBooleanTrueRep, and dfdl:textBooleanFalseRep. Property ignoreCase plays no part when comparing an element value with an XSD enum facet, matching an element value to an XSD pattern facet, or comparing an element value with the XSD fixed property. It is therefore not used by validation (when validation is enabled), nor by the dfdl:checkConstraints function. On unparsing always use the delimiters or value as specified. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
encodingErrorPolicy	Enum Valid values are 'error' or 'replace'. This property applies whenever dfdl:encoding is applicable. This property provides control of how decoding and encoding errors are handled when converting the data to text, or text to data. This includes converting when scanning for delimiters, matching regular expression length or test patterns, matching textual data type representation patterns against the data, and of course isolating the text content that becomes the value of an element (parsing) or constructing the content from the value (unparsing). When parsing, an error can occur when decoding characters from their encoded form into the DFDL Infoset character set (ISO10646). This can occur due to invalid byte sequences, or not enough bytes found to make up the full encoding of a character. If 'replace', then the Unicode replacement character (U+FFFD) is substituted for the offending errors, one replacement character for any incorrect fragment of an encoding. If 'error' then a Processing Error occurs. When unparsing, the errors that can occur when encoding characters from Unicode/ISO 10646 into the specified encoding include when no mapping is provided by the encoding character set specification and when there is not enough space to output the entire encoding of the character (e.g., need 2 bytes for a 2-byte character codepoint, but only 1 byte remains in the available length.) If 'replace' then encoding-specific replacement/substitution character is output. It is a Processing Error if no such character is defined, and it is a Processing Error if there is any error when attempting to output the replacement (such as not enough room for the representation of the entire encoding of the replacement character). If 'error' then a Processing Error occurs. See Section 11.2 Character Encoding and Decoding Errors for further details. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group

Table 13 Properties Common to both Content and Framing

11.1 Unicode Byte Order Mark (BOM)

DFDL does not provide any special treatment of Unicode Byte-Order Marks. They are treated as a Unicode ZWNBS character.

11.2 Character Encoding and Decoding Errors

When parsing, these are the errors that can occur when decoding characters into Unicode/ISO 10646.

1. The data is broken - invalid bit/byte sequences are found which do not match the definition of a character for the encoding.
2. Not enough data is found to make up the entire encoding of a character. That is, a fragment of a valid encoding is found.

When unparsing, these are the errors that can occur when encoding characters from Unicode/ISO 10646 into the specified encoding.

1. No mapping provided by the encoding specification.
2. Not enough room to output the entire encoding of the character (e.g., need 3 bytes for a character encoding that uses 3-bytes for that character, but only 1 byte remains in the available length.

The subsections below describe how these errors are handled.

11.2.1 **Property dfdl:encodingErrorPolicy**

The property dfdl:encodingErrorPolicy has two possible values: 'error' and 'replace'.

11.2.1.1 **dfdl:encodingErrorPolicy 'error'**

If 'error', then any error when decoding characters while parsing causes a Processing Error. For unparsing, any error when encoding characters causes a Processing Error.

When parsing, it does not matter if this happens when scanning for delimiters, matching a regular expression, matching a literal nil value, or constructing the value of a textual element.

There is one exception. When dfdl:lengthUnits is 'bytes', the 'not enough data' decoding error is ignored, and the data making up the fragment character is skipped over. Symmetrically, when unparsing the 'not enough room' encoding error is ignored and the left-over bytes are filled with the dfdl:fillByte.

Detection of character set decoding errors is often implementation-dependent because DFDL Implementations are free to optimize processing speed by skipping character decoding or encoding whenever possible. For example: when character set encodings are fixed-width, it is possible to determine lengths in bytes or bits from the length in characters by multiplying the length value by the character width, without having to decode any characters.

When parsing, character decoding errors MUST be detected when

a) the decoding results in a character being placed into the DFDL Infoset
b) the decoding is necessary to identify a delimiter
c) the decoding is necessary to determine a match or non-match of a regular expression in a dfdl:assert or dfdl:discriminator with testKind='pattern'.

When unparsing, character encoding errors MUST be detected when

d) an unmapped character appears in the Infoset value of an element.

In all other cases, character set decoding and encoding errors MAY not be detected.

Implementations MAY pre-decode a limited number of characters for efficiency; however, such implementation-dependent pre-decoding can cause parse errors to be detected in some implementations of DFDL that are not detected by others.

Schema authors are advised not to rely on decoding errors for backtracking to control the behavior of the parser.

11.2.1.2 **dfdl:encodingErrorPolicy 'replace' for parsing**

If 'replace' then any error when decoding characters results in the insertion of the Unicode Replacement Character (U+FFFD) as the replacement for that error.

It does not matter if this error and replacement happens when scanning for delimiters, matching a regular expression, matching a literal nil value, or constructing the value of a textual element.

There is one exception. When dfdl:lengthUnits is 'bytes', the 'not enough data' decoding error is ignored, no replacement character is created. The data making up the fragment character is skipped over. (It is filled with the dfdl:fillByte when unparsing.)

Note that the "." wildcard in regular expressions matches the Unicode Replacement Character, so ".*" and ".+" regular expressions can potentially cause very large matches (up to the entire data stream) to occur when data contains errors and dfdl:encodingErrorPolicy 'replace'. DFDL Schema authors are advised that bounded length negated regular expressions can help in this case. E.g., "[^\uFFFD]{0,50}" says to match any character (excluding the Unicode Replacement Character), but only up to length 50.

It is also worth noting that the Unicode Replacement Character can appear in data as an ordinary character, and this cannot be distinguished from the insertion of the Unicode Replacement Character due to a decoding error. This is likely to happen for data that is (a) initially parsed by a DFDL parser with dfdl:encodingErrorPolicy 'replace', and (b) which contains some decoding errors, but (c) is nevertheless successfully parsed, (d) is written back out to a file or other data repository, and (e) is parsed again. The written data has replaced data errors with the Unicode Replacement Character, and so if the data is parsed again, it no longer produces errors, but instead contains the Unicode Replacement Character as a regular character in the data.

If dfdl:lengthUnits is 'characters', then a Unicode Replacement Character counts as contributing a single character to the length.

If the data contains more than one adjacent decode error, then the specific number of Unicode Replacement Characters that are inserted as the replacement of these errors is implementation- dependent. That is, some implementations MAY view, for example, three consecutive erroneous bytes as three separate decode errors, others MAY view them as a single or two decode errors. All implementations MUST, however, insert some number of Unicode Replacement Characters, and then continue to decode characters following the erroneous data.

The trimming of pad characters always happens after Unicode Replacement Characters have been inserted into the data.

11.2.1.3 dfdl:encodingErrorPolicy 'replace' for unparsing

For unparsing, each encoding has a replacement/substitution character specified by the ICU. This character is substituted for the unmapped character or the character that has too large an encoding to fit in the available space.

There is one exception. When dfdl:lengthUnits is 'bytes', the 'not enough room' encoding error is ignored. The left-over bytes are filled with the dfdl:fillByte (they are skipped when parsing.)

The definitions of these substitution characters can be conveniently found for many encodings in the ICU Converter Explorer (http://demo.icu-project.org/icu-bin/convexp).

An encoding error is a Processing Error if the encoding does not provide a substitution/replacement character definition. (This would be rare but can occur if a DFDL implementation allows many encodings beyond the minimum set.)

11.2.2 Unicode UTF-16 Decoding/Encoding Non-Errors

The following specific situations involving encodings UTF-16, UTF-16LE, and UTF-16BE when dfdl:utf16Width "fixed", and they do not cause a decoding or encoding error.

- unpaired surrogate codepoint
- out-of-order surrogate codepoint pair
- surrogate codepoint pair is encountered

In all these cases the code-point(s) becomes a character code in the DFDL Information Item for the string.

11.2.3 Preserving Data Containing Decoding Errors

There can be situations where data wants to be preserved exactly even if it contains errors.

It is suggested that if a DFDL schema author wants to preserve information containing data where the encodings have these kinds of errors, that they model such data as xs:hexBinary, or as xs:string but using an encoding such as iso-8859-1 which preserves all bytes.

11.3 Byte Order and Bit Order

Byte order and bit order are separate concepts. However, of the possible combinations, only the following are allowed:

1. 'bigEndian' with 'mostSignificantBitFirst'
2. 'littleEndian' with 'mostSignificantBitFirst'
3. 'littleEndian' with 'leastSignificantBitFirst' [34]

Other combinations MUST produce Schema Definition Errors.

11.4 dfdl:bitOrder Example

Consider a structure of 4 logical elements. The total length is 16 bits.

Assume the lengths here are measured in bits (dfdl:lengthUnits[35] is 'bits'), and that these are binary integers (dfdl:representation is 'binary', dfdl:binaryNumberRep[36] is 'binary'):

```
<element name="A" type="xs:int" dfdl:length="3"/> <!-- having value 3 -->
<element name="B" type="xs:int" dfdl:length="7"/> <!-- having value 9 -->
<element name="C" type="xs:int" dfdl:length="4"/> <!-- having value 5 -->
<element name="D" type="xs:int" dfdl:length="2"/> <!-- having value 1 -->
```

[34] Used by data format MIL-STD-2045

[35] For dfdl:lengthUnits, see Section 12.3 Properties for Specifying Lengths.

[36] For dfdl:binaryNumberRep, see Section 13.7 Properties Specific to Number with Binary Representation.

The above are colorized to highlight the corresponding bits in the data below.

In a format where dfdl:bitOrder is 'mostSignificantBitFirst':

```
              01100010 01010101
              AAABBBBB BBCCCCDD
Significance  M      L M      L
Bit Position  12345678 12345678
Byte Position ----1--- ----2---
```

As presented here, the bits corresponding to each element appear left to right, and all bits for an individual element are adjacent. Within the bits of an individual element the most significant bit is on the left, least significant on the right, consistent with the way the bytes themselves are presented.

In contrast, in a format where dfdl:bitOrder is 'leastSignificantBitFirst':

```
              01001011 01010100
              BBBBBAAA DDCCCCBB
Significance  M      L M      L
Bit Position  87654321 87654321
Byte Position ----1--- ----2---
```

In the above presentation note how the bits of the element 'B' do not appear adjacent to each other. The most significant bits of byte N are adjacent to the least significant bits of byte N+1.

11.4.1 Example Using Right-to-Left Display for 'leastSignificantBitFirst'

When working exclusively with data having dfdl:bitOrder 'leastSignificantBitFirst', it is useful to present data with bytes Right to Left. That is, with the bytes starting at byte 1 on the right and increasing to the left.

```
              01010100 01001011
              DDCCCCBB BBBBBAAA
Significance  M      L M      L
Bit Position  87654321 87654321
Byte Position ----2--- ----1---
```

With this reorientation, the bits of the element 'B' are once again displayed adjacently. Within the bits of an individual element the most significant bit is on the left, least significant on the right, consistent with the way the bytes themselves are presented.

Often the specification documents for data formats using least-significant-bit-first bit order describe data using this Right-to-Left presentation style.

11.4.2 dfdl:bitOrder and Grammar Regions

When any grammar region appears before (to the left of) or after (to the right of) another grammar region in the grammar rules of Section 9.2, and the boundary between the two falls within a byte rather than on a byte boundary, then the dfdl:bitOrder determines which bits are occupied by the regions.

In general, the notion of *before* means occupying lower-numbered bit positions, and the bit positions are numbered according to dfdl:bitOrder. Hence, when dfdl:bitOrder is 'mostSignificantBitFirst', grammar regions that are before, occupy more-significant bits, and when dfdl:bitOrder is 'leastSignificantBitFirst', grammar regions that are before occupy less-significant bits.

12 Framing

Several properties are common across the various framing styles or are used to distinguish them. Generally, these have to do with position and length for text, bit fields, or opaque data.

12.1 Aligned Data

Alignment properties control the leading alignment and trailing alignment regions. That is, the LeadingAlignment and TrailingAlignment regions of the data syntax grammar (in Section 9.2).

When the alignment properties are applied to an array element, the properties are applied to each occurrence of the element; that is, not only to the first occurrence.

The following properties are used to define alignment rules.

Property Name	Description
alignment	Non-negative Integer or 'implicit' A non-negative number that gives the alignment required for the beginning of the item. If alignment is needed then the size of the ***AlignmentFill*** grammar region is non-zero if the item must be aligned to a boundary. 'implicit' specifies that the natural alignment for the representation type is used. See the table of implicit alignments Table 15 Implicit Alignment in bits for simple elements. The 'implicit' alignment of a complex element is the alignment of its model group. The 'implicit' alignment of a model group is always 1. If alignment is 'implicit' then dfdl:alignmentUnits is ignored. For textual data, minimum alignment is mandated by the character-set encoding, and this property must be 'implicit' or set to a multiple of the character-set's mandatory alignment. See Section 12.1.2. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
alignmentUnits	Enum Valid values are 'bits' or 'bytes' Scales the alignment so alignment can be specified in either units of bits or units of bytes. Only used when dfdl:alignment not 'implicit' Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
fillByte	DFDL String Literal A single byte specified as a DFDL byte value entity or a single character. If a character is specified, it must be a single-byte character in the applicable encoding. Used on unparsing to fill empty space such as between two aligned elements. Used to fill these regions specified in the grammar: ***RightFill***, ***ElementUnused***, ***ChoiceUnused, LeadingSkip***, ***AlignmentFill***, and ***TrailingSkip***. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
leadingSkip	Non-negative Integer A non-negative number of bytes or bits, depending on dfdl:alignmentUnits, to skip before alignment is applied. Gives the size of the grammar region having the same name. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
trailingSkip	Non-negative Integer A non-negative number of bytes or bits, depending on dfdl:alignmentUnits, to skip after the element, but before considering the alignment of the next element. Gives the size of the grammar region having the same name. If dfdl:trailingSkip is specified when dfdl:lengthKind is 'delimited' then a dfdl:terminator must be specified. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group

Table 14 Aligned Data Properties

There are two properties which control the data alignment by controlling the length of the ***AlignmentFill*** region

- alignment - an integer 1 or greater
- alignmentUnits - bits or bytes

An element's representation is aligned to N units if P is the first position in the representation and P mod N = 1. When parsing, the position of the first unit of the data stream is 1.

For example, if dfdl:alignment is 4, and dfdl:alignmentUnits is 'bytes', then the element's representation must begin at 1 or 1 plus a multiple of 4 bytes. That is, 1, 5, 9, 13, 17 and so on.

The length of the **AlignmentFill** region is measured in bits. If alignmentUnits is 'bytes' then the processor multiplies the alignment value by 8 to get the bit alignment, If the position in the data stream of the start of the **AlignmentFill** region is bit position N, then the length of the **AlignmentFill** region is the smallest non-negative integer L such that (L + N) mod B = 1. The position of the first bit of the aligned component is P = L + N.

The ***LeadingSkip*** and ***TrailingSkip*** regions length are controlled by two properties of corresponding names and the dfdl:alignmentUnits property.

12.1.1 Implicit Alignment

When dfdl:alignment is 'implicit' the following alignment values are applied for each logical type.

<table>
<tr><th rowspan="2">Type</th><th colspan="3">Alignment</th></tr>
<tr><th>text</th><th colspan="2">binary</th></tr>
<tr><td>String</td><td rowspan="14">Encoding Specific (usually 8 bits, with exceptions: See Section 12.1.2)</td><td colspan="2">Not applicable</td></tr>
<tr><td>Float</td><td colspan="2">32</td></tr>
<tr><td>Double</td><td colspan="2">64</td></tr>
<tr><td>Decimal, Integer, nonNegativeInteger</td><td>Packed decimals: 8</td><td>binary: 8</td></tr>
<tr><td>Long, UnsignedLong</td><td rowspan="9"></td><td>binary: 64</td></tr>
<tr><td>Int, UnsignedInt</td><td>binary: 32</td></tr>
<tr><td>Short, UnsignedShort</td><td>binary: 16</td></tr>
<tr><td>Byte, UnsignedByte</td><td>binary: 8</td></tr>
<tr><td>DateTime</td><td>binarySeconds: 32, binaryMilliseconds:64</td></tr>
<tr><td>Date</td><td>binarySeconds: 32, binaryMilliseconds:64</td></tr>
<tr><td>Time</td><td>binarySeconds: 32, binaryMilliseconds:64</td></tr>
<tr><td>Boolean</td><td colspan="2">32</td></tr>
<tr><td>HexBinary</td><td>Not applicable</td><td colspan="2">8</td></tr>
</table>

Table 15 Implicit Alignment in bits

Note: The above table specifies the implicit alignment in bits, but this does not imply that dfdl:alignmentUnits 'bits' can be specified for all simple types. Rather, dfdl:alignmentUnits and dfdl:lengthUnits are independent and have their own rules for when they are applicable.

12.1.2 Mandatory Alignment for Textual Data

Textual Data – This term is used to describe data of type xs:string, data with dfdl:representation "text", as well as data being matched to delimiters (parsing) or output as delimiters (unparsing), and data being matched to regular expressions (parsing only - as in a dfdl:assert with testKind 'pattern', or an element with dfdl:lengthKind 'pattern').

Textual data has mandatory alignment that is character-set-encoding dependent. That is, these mandates come from the character set encoding specified by the dfdl:encoding property.

When processing textual data, it is a Schema Definition Error if the dfdl:alignment and dfdl:alignmentUnits properties are used to specify alignment that is not a multiple of the encoding-specified mandatory alignment.

If the data is not aligned to the proper boundary for the encoding when textual data is processed, then bits are skipped (parsing) or filled from dfdl:fillByte (unparsing) to achieve the mandatory alignment.

All required character set encodings in DFDL have 8-bit/1-byte alignment.

DFDL standard encodings specify their alignment. See Section 33 Appendix D: DFDL Standard Encodings.

Some implementations MAY include additional implementation-defined encodings which have other alignments.

Note the 16-bit and 32-bit Unicode character set encodings UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE, all have 8-bit/1-byte alignment.

12.1.3 Mandatory Alignment for Packed Decimal Data

Packed decimal data is data with dfdl:binaryNumberRep[37] values of 'packed', 'ibm4690Packed' or 'bcd'. This representation stores a decimal digit in a 4 bit nibble. These nibbles must have a multiple of 4-bit alignment. It is a Schema Definition Error otherwise.

12.1.4 Example: AlignmentFill

When dfdl:alignmentUnits is 'bits', and the dfdl:alignment is not a multiple of 8, then the dfdl:bitOrder property affects the alignment by controlling which bits are skipped as part of the grammar ***AlignmentFill*** region.

In general, the ***AlignmentFill*** region is *before* the regions it is aligning, and within a byte, the meaning of *'before'* is interpreted with respect to the dfdl:bitOrder.

When dfdl:bitOrder is 'mostSignificantBitFirst', then bits with more significance are before bits with less significance, so the ***AlignmentFill*** region occupies the most significant bits of the byte.

When dfdl:bitOrder is 'leastSignificantBitFirst', then bits with less significance are before bits with more significance, so the ***AlignmentFill*** region occupies the least significant bits of the byte.

Consider a structure of 2 logical elements. Assume the length and alignment units are bits. (dfdl:lengthUnits='bits', dfdl:alignmentUnits='bits'), and that the data is binary with twos-complement binary integers (dfdl:representation='binary', dfdl:binaryNumberRep='binary'), and assume the data is at the beginning of the data stream.

```
<element name="A" type="xs:int" dfdl:length="2" dfdl:alignment='8'/>
<!-- having value 1 -->
<element name="B" type="xs:int" dfdl:length="4" dfdl:alignment='4'/>
<!-- having value 5 -->
```

The above are colorized to highlight the corresponding bits in the data below. The total length due to the alignment region appearing before element 'B' is 8 bits.

In a format where dfdl:bitOrder is 'mostSignificantBitFirst' the data can be visualized as:

```
              01000101
              AAxxBBBB
Significance  M      L
Bit Position  12345678
```

In the above, the ***AlignmentFill*** region is marked with 'x' characters and contains all 0 bit values.

In a format where dfdl:bitOrder is 'leastSignificantBitFirst' the presentation is different:

```
              01010001
              BBBBxxAA
Significance  M      L
Bit Position  87654321
```

In the above the ***AlignmentFill*** region still appears before element 'B', and in this case that is in less significant bits of the byte than the bits of content of element 'B', and these bits are displayed to the right of the bits of element 'B'.

[37] For dfdl:binaryNumberRep, see Section 13.7 Properties Specific to Number with Binary Representation.

12.2 Properties for Specifying Delimiters

The following properties apply to all objects that use text delimiters to delimit, that is, to initiate and/or terminate data. Delimiters can apply to binary data; however, they are most often called 'text' delimiters because the concept is much more commonly used for textual data formats.

When parsing, there can be multiple delimiter candidates to be matched against the data stream. The matching is performed in a *longest-match preferred* manner. That is, each of the delimiter candidates is matched against the data, taking the longest match possible for that candidate. Then across all the delimiter candidates, the one with the longest match is the one that is selected as having been found. Once a matching delimiter is found, no other matches are subsequently attempted (i.e., there is no backtracking to try shorter matches.) Additional details on the matching of DFDL String Literals are given in Appendix C: Processing of DFDL String literals.

Property Name	Description
initiator	List of DFDL String Literals or DFDL Expression Specifies an ordered whitespace separated list of alternative DFDL String Literals one of which marks the beginning of the element or group of elements. This property can be computed by way of an expression which returns a string containing a whitespace separated list of DFDL String Literals. The expression must not contain forward references to elements which have not yet been processed. It is not permitted for an expression to return an empty string or a string containing only whitespace. That is a Schema Definition Error. Each string literal in the list, whether apparent in the schema, or returned as the value of an expression, is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed. • DFDL Byte Value entities (%#rXX;) are allowed. • DFDL Character Classes NL, WSP, WSP+, WSP*, and ES are allowed. • If the ES entity or the WSP* entity appear alone as one of the string literals in the list, then dfdl:initiatedContent must be "no". This restriction ensures that when dfdl:initiatedContent is 'yes' that the initiator cannot match zero-length data. If the above rules are not followed it is a Schema Definition Error. The ***Initiator*** region contains one of the initiator strings defined by dfdl:initiator. When parsing, once a matching initiator is found, no other matches are subsequently attempted (i.e., there is no backtracking). When an initiator is specified, it is a Processing Error if the component is required and one of the values is not found. If dfdl:initiator is "" (the empty string), that is the way a DFDL schema expresses a format which does not use initiators. Hence, the ***Initiator*** region is of length zero. On unparsing the first initiator in the list is automatically inserted into the ***Initiator*** region. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group

terminator	List of DFDL String Literals or DFDL Expression Specifies an ordered whitespace separated list of alternative text strings that one of which marks the end of an element or group of elements. The strings MUST be searched for in the longest first order. This property can be computed by way of an expression which returns a string of whitespace separated list of values. The expression must not contain forward references to elements which have not yet been processed. This property can be used to determine the length of an element as described in Section 12.3.2 dfdl:lengthKind 'delimited'. Each string literal in the list, whether apparent in the schema, or returned as the value of an expression, is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed. • DFDL Byte Value entities (%#rXX;) are allowed. • DFDL Character Classes NL, WSP, WSP+, WSP*, and ES are allowed. • Neither the ES entity nor the WSP* entity may appear on their own as one of the string literals in the list when the parser is determining the length of a component by scanning for delimiters. If the above rules are not followed it is a Schema Definition Error. The ***Terminator*** grammar region contains one of the terminator strings defined by dfdl:terminator. If dfdl:terminator is "" (the empty string), that is the way a DFDL schema expresses a format which does not use terminators. Hence, the ***Terminator*** region is of length zero. It is not permitted for an expression to return an empty string, that is a Schema Definition Error. When parsing, once a matching terminator is found, no other matches are subsequently attempted (i.e., there is no backtracking). When a terminator is expected it is a Processing Error if no matching terminator is found. However, if dfdl:documentFinalTerminatorCanBeMissing is specified then it is not an error if the last terminator in the data stream is not found. On unparsing the first terminator in the list is automatically inserted in the Terminator region. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
emptyValueDelimiterPolicy	Enum Valid values are 'none', 'initiator', 'terminator' or 'both' Indicates that when an element in the data stream is empty, which of initiator, terminator, both, or neither must be present. Ignored if both dfdl:initiator and dfdl:terminator are "" (empty string). 'initiator' indicates that, on parsing, if the content region (which can be either the SimpleContent region or the ComplexContent region defined in Section 9.2) is empty then the dfdl:initiator

	must be present. It also indicates that on unparsing when the content region is empty that the dfdl:initiator is output. 'terminator' indicates that, on parsing, if the content region is empty then the dfdl:terminator must be present. It also indicates that on unparsing when the content region is empty the dfdl:terminator is output. 'both' indicates that, on parsing, if the content region is empty both the dfdl:initiator and dfdl:terminator must be present. On unparsing when the content region is empty the dfdl:initiator followed by the dfdl:terminator is output. 'none' indicates that if the content region is empty neither the dfdl:initiator or dfdl:terminator must be present. On unparsing when the content region is empty nothing is output. It is a Schema Definition Error if dfdl:emptyValueDelimiterPolicy set to 'none' or 'terminator' when the parent group has dfdl:initiatedContent 'yes'. This property plays an important role in establishing empty representation. See 9.2.2 Empty Representation for details. This property is ignored if the element is fixed-length and length is not zero (as no empty representation is possible). The value of dfdl:emptyValueDelimiterPolicy MUST only be checked if there is a dfdl:initiator or dfdl:terminator in scope. If so, and dfdl:emptyValueDelimiterPolicy is not set, it is a Schema Definition Error. If dfdl:initiator is not "" and dfdl:terminator is "" and dfdl:emptyValueDelimiterPolicy is 'terminator' it is a Schema Definition Error. If dfdl:terminator is not "" and dfdl:initiator is "" and dfdl:emptyValueDelimiterPolicy is 'initiator' it is a Schema Definition Error. It is not a Schema Definition Error if dfdl:emptyValueDelimiterPolicy is 'both' and one or both of dfdl:initiator and dfdl:terminator is "". This is to accommodate the common use of setting 'both' as a schema-wide setting. It is a Schema Definition Error if dfdl:emptyValueDelimiterPolicy is in effect and is set to 'none' or 'terminator' when the parent xs:sequence has dfdl:initiatedContent 'yes'. Annotation: dfdl:element, dfdl:simpleType
documentFinalTerminatorCanBeMissing	Enum Valid values are 'yes', 'no' When the dfdl:documentFinalTerminatorCanBeMissing property is true, then when an element is the last element in the data stream, then on parsing, it is not an error if the terminator is not found, and the terminator is considered to be logically present for the purposes of establishing representation, per Section 9.3.2. For example, if the data are in a file, and the format specifies lines terminated by the newline character (typically LF or CRLF), then if the last line is missing its newline, then this would normally be an error, but if dfdl:documentFinalTerminatorCanBeMissing is true, then this is not a Processing Error. On unparsing the terminator is always written out regardless of the state of this property. Annotation: dfdl:format (but applies to elements only)

outputNewLine	DFDL String Literal or DFDL Expression Specifies the character or characters that are used to replace the %NL; character class entity during unparse. (The %NL; entity is defined in Section 6.3.1.3 DFDL Character Class Entities in DFDL String Literals.) It is a Schema Definition Error if any of the characters are not in the set of characters allowed by the DFDL entity %NL; Only individual characters or the %CR;%LF; combination are allowed. It is a Schema Definition Error if the DFDL entity %NL; is specified This property can be computed by way of an expression which returns a DFDL string literal. The expression must not contain forward references to elements which have not yet been processed. Annotation: dfdl:element, dfdl:simpleType, dfdl:sequence, dfdl:choice, dfdl:group
emptyElementParsePolicy	Enum Valid values are "treatAsAbsent" or "treatAsEmpty". This property describes the behavior of the DFDL processor for occurrences of elements of any type that have the empty representation. When 'treatAsEmpty' if an occurrence of an element has the empty representation when parsed, the behaviour is as stated in Section 9 for an occurrence with empty representation. Consequently, default values or empty strings may be added to the Infoset. When 'treatAsAbsent' if an occurrence of an element has the empty representation when parsed, the behaviour is as stated in Section 9 for an absent occurrence. Consequently, default values or empty strings are never added to the Infoset. Annotation: dfdl:element, dfdl:simpleType

Table 16 Properties for Specifying Delimiters

12.3 Properties for Specifying Lengths

These properties are used to determine the content length of an element and apply to elements of all types (simple and complex).

Property Name	Description
lengthKind	Enum Controls how the content length of the component is determined. Valid values are: 'explicit', 'delimited', 'prefixed', 'implicit', 'pattern', 'endOfParent' A full description of each enumeration is given in the subsections of this section beginning with Section 12.3.1. 'explicit' means the length of the element is given by the dfdl:length property. 'delimited' means the element length is determined by scanning for a terminator or separator. 'prefixed' means the length of the element is given by an immediately preceding PrefixLength data region the format of which is specified using dfdl:prefixLengthType. 'implicit means the length is to be determined in terms of the type of the element and its schema-specified properties if any. 'pattern' means the length of the element is given by scanning for a regular expression specified using the dfdl:lengthPattern property.

	'endOfParent' means that the length extends to the end of the containing (parent) construct. Annotation: dfdl:element, dfdl:simpleType
lengthUnits	Enum Valid values 'bytes', 'characters', and 'bits'. Specifies the units to be used whenever a length is being used to extract or write data. Applicable when dfdl:lengthKind is 'explicit', 'implicit' (for xs:string and xs:hexBinary) or 'prefixed'. Usage is restricted as follows: • 'characters' may only be used for complex elements and simple elements with text representation. • 'bits' may only be used for xs:boolean, xs:byte, xs:short, xs:int, xs:long, xs:unsignedByte, xs:unsignedShort, xs:unsignedInt, and xs:unsignedLong simple types with binary representation, and for calendar (date and time) simple types with binary packed representation. • 'bytes' must be used for type xs:hexBinary and for types xs:float and xs:double with binary representation. 'bytes' may be used for any other type. Annotation: dfdl:element, dfdl:simpleType

Table 17 Properties for Specifying Length

12.3.1 dfdl:lengthKind 'explicit'

When dfdl:lengthKind is 'explicit' the length of the item is given by the dfdl:length property.

When the value of the dfdl:length property is a constant, it is used both when parsing and unparsing.

When unparsing an element with dfdl:lengthKind 'explicit' and where dfdl:length is an expression, then the data in the Infoset is treated as fixed-length and the dfdl:length property, whether literal constant or expression, is evaluated to provide the length to use.

When parsing and dfdl:lengthKind is 'explicit', delimiter scanning is turned off and in-scope delimiters are not looked for within or between elements.

Property Name	Description
length	Non-negative Integer or DFDL Expression. Only used when lengthKind is 'explicit'. Specifies the length of this element in units that are specified by the dfdl:lengthUnits property. This property can be computed by way of an expression which returns a non-negative integer. The expression must not contain forward references to elements which have not yet been processed. Annotation: dfdl:element, dfdl:simpleType

Table 18 The dfdl:length Property

When dfdl:lengthKind 'explicit', the method of extracting data is described in Section: 12.3.7 Elements of Specified Length

12.3.2 dfdl:lengthKind 'delimited'

On parsing, the length of an element with dfdl:lengthKind 'delimited' is determined by scanning the data stream for the delimiter.

The data stream is scanned for any of

- the element's terminator (if specified)
- an enclosing construct's separator or terminator
- the end of an enclosing element designated by its known length
- the end of the data stream

dfdl:lengthKind 'delimited' may be specified for

- elements of simple type with text representation

- elements of number or calendar (date and time) simple type with dfdl:representation 'binary' that have a packed decimal representation
- elements of type xs:hexBinary
- elements of complex type.

The rules for resolving ambiguity between delimiters are:

1. When two delimiters have a common prefix, the longest delimiter is tried first.
2. When two delimiters have the same length, but on different schema components, the innermost (most deeply nested) delimiter is tried first.
3. When the separator and terminator on a group have the same value, then at a point in the data where either the separator or terminator could be found, the separator is tried first. (Speculative execution may try the terminator subsequently).
4. If the length of the delimiters cannot be determined because character class entities of variable length are being used then the delimiters MUST each be matched against the data, and the longest matching delimiter is taken as the match for the delimiter.
5. Ties (same matched length) are broken by giving a separator priority over a terminator of a sequence, or by choosing the innermost, or first in schema order.

When unparsing a simple element with text representation, the length in the data stream is the length of the content region, padded to a minimum length if dfdl:textPadKind is ‘padChar’. For xs:string elements this length is the XSD minLength facet value, for the other types it is dfdl:textOutputMinLength property value.

When unparsing a simple element with binary representation, then for hexBinary the length is the number of bytes in the Infoset value padded to the XSD minLength facet value using dfdl:fillByte, and for the other types the length is the minimum number of bytes to represent the value and any sign.

When unparsing a complex element, the length is that of the ComplexContent region.

12.3.2.1 Non-Delimited Elements within Delimited Constructs

When a simple or complex element has a specified length, dfdl:lengthKind 'pattern', or dfdl:lengthKind 'endOfParent' then delimiter scanning is suspended for the duration of the processing of that element.

This allows formats to be parsed which are delimited but have nested elements which contain non-character data so long as that nested data can be isolated from the delimited data context surrounding it.

12.3.2.2 Delimited Binary Data

Formats involving binary data, most notably packed decimals, can use delimiter scanning but care must be taken that the delimiters cannot match data represented in these formats. In particular, the delimiters must be chosen with knowledge that BCD data can contain any byte both of whose nibbles are 0 to 9 (that is, excluding A to F). Packed data adds bytes with a sign indicator, that is, a nibble in the range A to F.

General binary data can contain any bit pattern whatsoever, so delimiter scanning for numbers and calendar types with dfdl:representation 'binary' is disallowed, with the specific exception of packed decimals. Delimiter scanning is also allowed for type xs:hexBinary.

Implementation Note: Scanning for delimiters when data is binary, or when using byte-value (aka raw byte) entities in delimiters means that a simple character-based delimiter scanner IS NOT sufficient, as the delimiter may not be representable as characters.

12.3.3 dfdl:lengthKind 'implicit'

When dfdl:lengthKind is 'implicit', the length is determined in terms of the type of the element and its schema-specified properties.

For complex elements, 'implicit' means the length is determined by the combined lengths of the contained children, that is the ComplexValue region, and the ElementUnused region is of size 0. However, note that alignment regions inside the contained children within the ComplexValue region may be of different lengths depending on the ComplexValue's starting position alignment.

For simple elements the length is fixed and is given in Table 19 Length in Bits for SimpleTypes when dfdl:lengthKind is 'implicit' .

Type	Length	
	text	binary
String	The XSD maxLength facet gives length in characters,	Not applicable

	but this is also the length in bytes. (See note below: character set encoding must be single-byte.) Multiply by 8 to get number of bits.		
Float	Not allowed	32 bits	
Double	Not allowed	64 bits	
Decimal, Integer, nonNegativeInteger	Not allowed	packed decimal: Not allowed	binary: Not allowed
Long, UnsignedLong	Not allowed		binary: 64 bits
Int, UnsignedInt	Not allowed		binary: 32 bits
Short, UnsignedShort	Not allowed		binary: 16 bits
Byte, UnsignedByte	Not allowed		binary: 8 bits
DateTime	Not allowed		binarySeconds: 32 bits, binaryMilliseconds: 64 bits
Date	Not allowed		binarySeconds: Not allowed, binaryMilliseconds: Not allowed
Time	Not allowed		binarySeconds: Not allowed, binaryMilliseconds: Not allowed
Boolean	Length of longest of dfdl:textBooleanTrueRep and dfdl:textBooleanFalseRep values	32 bits	
HexBinary	Not applicable	The XSD maxLength facet gives the length in bytes. Multiply by 8 to convert to number of bits.	

Table 19 Length in Bits for SimpleTypes when dfdl:lengthKind is 'implicit'

- 'Not Allowed' means that there is no implicit length for the combination of simple type and representation, and it is a Schema Definition Error if dfdl:lengthKind 'implicit' is specified.
- packed decimal means dfdl:binaryNumberRep is 'packed', 'bcd', or 'ibm4690Packed'
- binary means dfdl:binaryNumberRep is 'binary'
- binarySeconds means dfdl:binaryCalendarRep is 'binarySeconds'
- binaryMilliseconds means dfdl:binaryCalendarRep is 'binaryMilliseconds'.

When dfdl:lengthKind is 'implicit', the method of extracting data is described in Section 12.3.7 Elements of Specified Length

It is a Schema Definition Error if type is xs:string and dfdl:lengthKind is 'implicit' and dfdl:lengthUnits is 'bytes' and encoding is not an SBCS (exactly 1 byte per character code) encoding. This prevents a scenario where validation against the XSD maxLength facet is in characters but parsing and unparsing using the XSD maxLength facet is in bytes.

12.3.4 **dfdl:lengthKind 'prefixed'**

When dfdl:lengthKind is 'prefixed' the length of the element is given by the integer value of the PrefixLength region specified using dfdl:prefixLengthType. The property dfdl:prefixIncludesPrefixLength also can be used to adjust the length appropriately.

When dfdl:lengthKind is 'prefixed' the method of extracting data is described in Section 12.3.7 Elements of Specified Length

When dfdl:lengthKind is 'prefixed', delimiter scanning is turned off and in-scope delimiters are not looked for within or between elements.

Property Name	Description
prefixIncludesPrefixLength	Enum Valid values are 'yes', 'no' Specifies whether the length given by a prefix includes the length of the prefix as well as the length of the content region which can be either the SimpleContent region or the ComplexContent region defined in Section 9.2 DFDL Data Syntax Grammar. Used only when dfdl:lengthKind 'prefixed'. Annotation: dfdl:element, dfdl:simpleType
prefixLengthType	QName Name of a simple type derived from xs:integer or any subtype of it. This type, with its DFDL annotations specifies the representation of the length prefix, which is in the PrefixLength region. It is a Schema Definition Error if the xs:simpleType or any base type thereof specifies any of: • dfdl:lengthKind 'delimited', 'endOfParent', or 'pattern' • dfdl:lengthKind 'explicit' where length is an expression • dfdl:outputValueCalc • dfdl:initiator or dfdl:terminator other than empty string • dfdl:alignment other than '1' • dfdl:leadingSkip or dfdl:trailingSkip other than '0'. • dfdl:assert, dfdl:discriminator, or dfdl:setVariable If the xs:simpleType is constrained by facets (minInclusive, maxInclusive, minExclusive, maxExclusive) these constraints are always checked, both when parsing and unparsing. It is a Processing Error if the value of the xs:simpleType does not conform to the facet constraints. It is a Processing Error if the value of the xs:simpleType is less than zero. Annotation: dfdl:element, dfdl:simpleType

Table 20 Properties for dfdl:lengthKind 'prefixed'

The representation of the element is in two parts.

1. The 'prefix length' is an integer which specifies the length of the element's content. The representation of the length prefix is described by a simple type which is identified using the dfdl:prefixLengthType property.
2. The content of the element.

When parsing, the length of the element's content is obtained by parsing the simple type specified by dfdl:prefixLengthType to obtain an integer value. Note that all required properties must be present on the specified simple type or defaulted because there is no element declaration to supply any missing required properties.

If the dfdl:prefixIncludesPrefixLength property is 'yes' then the length of the element's content is the value of the prefix length minus the length of the content of the prefix length.

If the prefix type is dfdl:lengthKind 'implicit' or 'explicit' then the dfdl:lengthUnits properties of both the prefix type and the element must be the same.

The DFDL properties that specify the format of the prefix come from annotations directly on the dfdl:prefixLengthType's type definition, and from the default format annotation for the schema document containing the definition of that type. If the using-element resides in a separate schema, the simple type does not pick up values from the element's schema's default dfdl:format annotation.

When unparsing, the length of the element's content region can be determined first as described below. Then the value of the prefix length MUST be adjusted based on the value of the dfdl:prefixIncludesPrefixLength property.

Then the prefix length can be written to the data stream using the properties on the dfdl:prefixLengthType, and finally the element's content can be written to the data stream.

Consider this example:

```
<xs:element name="myString" type="xs:string"
                    dfdl:lengthKind="prefixed"
                    dfdl:prefixIncludesPrefixLength="no"
                    dfdl:prefixLengthType="packed3"/>

<xs:simpleType name="packed3"
            dfdl:representation="binary"
            dfdl:binaryNumberRep="packed"
            dfdl:lengthKind="explicit"
            dfdl:length="2" >
  <xs:restriction base="integer" />
</xs:simpleType>
```

In the above, the string has a prefix length of type 'packed3' containing 3 packed decimal digits.

The property dfdl:prefixIncludesPrefixLength is an enumeration which allows the length computation to be varied to include or exclude the length of the prefix element itself.

The prefix length's value contains the length measured in units given by dfdl:lengthUnits.

When parsing, if the dfdl:lengthUnits are bits, then any number of bits can be in the representation. However, the same is not true when unparsing. The DFDL Infoset does not store the number of bits in a number, so the number of bits is always a multiple of 8 bits.

When unparsing, the value of the prefix is computed automatically by obtaining the length of the element's content.

For a simple element with text representation, the length is computed as for dfdl:lengthKind 'delimited'.

For a simple element with binary representation, the length is given in the table below.

For a complex element, the length is that of the ComplexContent region.

<table>
<tr><th>Type</th><th colspan="2">Length</th></tr>
<tr><td>String</td><td colspan="2">Not applicable</td></tr>
<tr><td>Float</td><td colspan="2">32</td></tr>
<tr><td>Double</td><td colspan="2">64</td></tr>
<tr><td>Decimal, Integer, NonNegativeInteger</td><td colspan="2">Compute the minimum number of bytes to represent the value (per dfdl:binaryNumberRep) and sign (if applicable). Multiply by 8 for number of bits.</td></tr>
<tr><td>Long, UnsignedLong</td><td rowspan="8">packed decimal: as Decimal</td><td>binary: 64</td></tr>
<tr><td>Int, UnsignedInt</td><td>binary: 32</td></tr>
<tr><td>Short, UnsignedShort</td><td>binary: 16</td></tr>
<tr><td>Byte, UnsignedByte</td><td>binary: 8</td></tr>
<tr><td>DateTime</td><td>binarySeconds: 32, binaryMilliseconds:64</td></tr>
<tr><td>Date</td><td>binarySeconds: Not allowed, binaryMilliseconds: Not allowed</td></tr>
<tr><td>Time</td><td>binarySeconds: Not allowed, binaryMilliseconds: Not allowed</td></tr>
<tr><td>Boolean</td><td colspan="2">32</td></tr>
<tr><td>HexBinary</td><td colspan="2">Compute the number of bytes in the Infoset value padded to the value of the XSD minLength facet (which gives minimum length in bytes) using dfdl:fillByte if necessary. This gives the unparse length in bytes. Multiply by 8 for the number of bits.</td></tr>
</table>

Table 21 Unparse Lengths (in Bits) for Binary Data with dfdl:lengthKind 'prefixed'

12.3.4.1 Nested Prefix Lengths[38]

It is possible for a prefix length, as specified by dfdl:prefixLengthType, to itself have a prefix length . That is, an element can have a prefix length, which defines a PrefixLength region (see Section 9.2 DFDL Data Syntax Grammar), and the PrefixLength region can itself have a type which specifies a prefix length, thereby defining a PrefixPrefixLength region.

It is a Schema Definition Error if the type associated with the PrefixPrefixLength is the same as the type associated with the PrefixLength. It is a Schema Definition Error if a PrefixPrefixLength region has a type which specifies a prefix length, that is, nesting of prefix lengths must not exceed a depth of 1. It is a Processing Error if a PrefixPrefixLength region has zero length.

12.3.5 dfdl:lengthKind 'pattern'

The dfdl:lengthKind 'pattern' means the length of the element is given by a regular expression specified using the dfdl:lengthPattern property. The DFDL processor scans the data stream to determine a string value that is the match to a regular expression. The pattern is only used on parsing.

When dfdl:lengthKind is 'pattern', delimiter scanning is turned off and in-scope delimiters are not looked for within or between elements.

Property Name	Description
lengthPattern	DFDL Regular Expression. Only used when lengthKind is 'pattern'. Specifies a regular expression that, on parsing, is executed against the data stream to determine the length of the element. The data stream beginning at the starting offset of the content region (which can be either the SimpleContent region or the ComplexContent region defined in Section 9.2 DFDL Data Syntax Grammar) of the element is interpreted as a stream of characters in the encoding of the element, and the regular expression contained in the dfdl:lengthPattern property is executed against that stream of characters. When the element is complex the encoding used is the dfdl:encoding of the complex element itself. It is a Schema Definition Error if there is no value for the dfdl:encoding property in scope. DFDL Escape Schemes (per dfdl:escapeSchemeRef) are not used when executing the regular expression. If the pattern matching of the regular expression reads data that cannot be decoded into characters of the current encoding, then the behavior is controlled by the dfdl:encodingErrorPolicy property. See dfdl:encodingErrorPolicy in Section 11 Properties Common to both Content and Framing. Annotation: dfdl:element, dfdl:simpleType

Table 22 The dfdl:lengthPattern Property

On unparsing the behavior is the same as for dfdl:lengthKind 'prefixed'.

When the DFDL regular expression is matched against data:

- The data is considered to be text in the character set encoding specified by the dfdl:encoding property, regardless of the actual representation of the element.
- The data is decoded from the specified encoding into Unicode before the actual matching takes place.
- If there is no match (i.e., the length of the data found to match the pattern is zero) it is not a Processing Error but instead it means the length is zero.

12.3.6 dfdl:lengthKind 'endOfParent'

The dfdl:lengthKind 'endOfParent' means that the element is terminated either by the end of the data stream, or the end of an enclosing complex element with dfdl:lengthKind ‘explicit’, ‘pattern’, ‘prefixed’ or ‘endOfParent’, or the end of an enclosing choice with dfdl:choiceLengthKind ‘explicit’. The ‘parent’ element or choice does not

[38]This feature allows DFDL to describe the needed “one more level” of prefix that is needed for modeling an ASN.1 format, but without the complexities of general recursion.

have to be the immediate enclosing component of the element, but there must be no other components defined between the element specifying dfdl:lengthKind 'endOfParent' and the end of the parent.

A convenient way of describing the parent is as a 'box', being defined as a portion of the data stream that has an established content length prior to the parsing of its children. If the parent is such a ‘box’ then the element specifying dfdl:lengthKind ‘endOfParent’ is the last element in the ‘box’ and its content extends to the end of the ‘box’.

A dfdl:lengthKind of 'endOfParent' can only be used on simple and complex elements in the following locations:

- When the immediate containing model group is a sequence, on the final element in the sequence
- When the immediate containing model group is a choice, on any element that is a branch of the choice
- A simple type or global element declaration referenced by one of the above.
- A global element declaration that is the document root.

It is a Schema Definition Error if:

- the element has a terminator.
- the element has dfdl:trailingSkip not equal to 0.
- the element has maxOccurs > 1.
- any other model-group is defined between this element and the end of the enclosing component.
- any other represented element is defined between this element and the end of the enclosing component.
- the parent is an element with dfdl:lengthKind 'implicit' or 'delimited'.
- the element has text representation, does not have a single-byte character set encoding, and the effective length units of the parent is not ‘characters’.
- The effective length units of the parent are:
    - dfdl:lengthUnits if parent is an element with dfdl:lengthKind ‘explicit’ or ‘prefixed’;
    - ‘characters’ if parent is an element with dfdl:lengthKind ‘pattern’;
    - ‘bytes’ if parent is a choice with dfdl:choiceLengthKind ‘explicit’;
    - ‘characters’ if the element is the document root;
    - the effective length units of the parent’s parent if parent is an element with dfdl:lengthKind ‘endOfParent’

If the element is in a sequence then it is a Schema Definition Error if:

- the dfdl:separatorPosition of the sequence is 'postfix'
- the dfdl:sequenceKind of the sequence is not 'ordered'
- the sequence has a terminator
- there are floating elements in the sequence
- the sequence has a non-zero dfdl:trailingSkip

If the element is in a choice where dfdl:choiceLengthKind is 'implicit' then it is a Schema Definition Error if:

- the choice has a terminator
- the choice has a non-zero dfdl:trailingSkip

A simple element must have one of:

- type xs:string
- dfdl:representation 'text'
- type xs:hexBinary
- dfdl:representation 'binary' and a packed decimal representation

A complex element can have dfdl:lengthKind 'endOfParent'. If so then its last child element can be any dfdl:lengthKind including 'endOfParent'.

The dfdl:lengthKind 'endOfParent' can also be used on the document root to allow the last element to consume the data up to the end of the data stream.

The use of dfdl:lengthKind ‘endOfParent’ is distinct from the situation where the length of the last element in the parent is known but is not sufficient to fill the parent. In the latter case the remaining data are ignored on parsing and filled with dfdl:fillByte on unparsing.

When parsing an element with dfdl:lengthKind ‘endOfParent’, delimiter scanning is turned off and in-scope terminating delimiters are not looked for within the element.

When unparsing an element with dfdl:lengthKind ‘endOfParent’, if the parent is a complex element with dfdl:lengthKind 'explicit' where dfdl:length is not an expression, or a choice with dfdl:choiceLengthKind

'explicit', then the element with dfdl:lengthKind 'endOfParent' is padded or filled in the usual manner to the required length, by completing the ***LeftPadding***, ***RightPad***, ***RightFill***, ***ElementUnused***, or ***ChoiceUnused*** regions of the data syntax grammar (Section 9.2) as appropriate.

12.3.7 Elements of Specified Length

An element has a specified length when dfdl:lengthKind is 'explicit', 'implicit' (simple type only) or 'prefixed'. The units that the length represents are specified by the dfdl:lengthUnits property except where noted in Section 12.3.3.

Using specified length, it is possible for an element to have content length longer than needed to represent just the data value. For example, a simple text element may be padded in the ***RightPadding*** region if the data is not long enough.

When an element has specified length but appears inside a complex type element having delimited length kind, delimiter scanning is turned off and in-scope delimiters are not looked for within or between elements.

An element of specified length with dfdl:lengthKind 'implicit' or 'explicit' where dfdl:length is not an expression has a known length when unparsing.

An element of specified length with dfdl:lengthKind 'prefixed' is considered to have a *variable* length when unparsing. Specifically, the processor automatically determines the value to store in the prefix, based on the length of the SimpleContent or ComplexContent regions, and the properties which modify the interpretation of the prefix length value, such as dfdl:prefixIncludesPrefixLength.

For dfdl:lengthKind 'explicit' (expression), whether parsing or unparsing the expression is evaluated to obtain the length. When unparsing the processor cannot automatically determine in what way the length information is to be stored as it comes from an expression that may access one or more elements and perform any calculation. Hence, normally the value of the element or elements involved in the length calculation would be computed using dfdl:outputValueCalc, using an expression that measures the length of the element by way of functions such as dfdl:contentLength or dfdl:valueLength.

When parsing, if the data stream ends without enough data to parse an element, that is, N bits are needed based on the dfdl:length, but only M < N bits are available, then it is a Processing Error.

If dfdl:lengthUnits is 'characters' then the length (in bits) of the content region (i.e., SimpleContent or ComplexContent defined in Section 9.2 DFDL Data Syntax Grammar) depends on the encoding of the characters.

- If the dfdl:encoding property specifies a fixed-width encoding then the content length is the character width (in bits) multiplied by the length.
- If the dfdl:encoding property specifies a variable-width encoding then the length depends on the actual characters in the element's value. The characters MUST be decoded one by one, adding up their widths (in bits), while counting to the specified length value.

For a simple element, dfdl:lengthUnits 'characters' may only be used for textual elements, it is a Schema Definition Error otherwise.

Some DFDL implementations MAY support character set encodings where the characters are not a multiple of 8-bits wide. Encodings which are 5, 6, 7, and 9 bits wide are rare, but do exist, so the overall length of the content region may not be a multiple of 8-bits wide.

12.3.7.1 Length of Simple Elements with Textual Representation

Textual data is defined to mean either data of type string or data where the dfdl:representation property is 'text'.

For a textual element, the dfdl:lengthUnits property can be either 'bytes' or 'characters'.

12.3.7.1.1 Text Length Specified in Bytes

If a textual element has dfdl:lengthUnits of 'bytes', and the dfdl:encoding is not SBCS, then it is possible for a partial character encoding to appear after the code units of the characters. In this case, the following rules apply:

- When parsing, as many characters as possible are extracted from the bytes of the simple content region. Any left-over bytes are skipped. (They are considered part of the grammar ***RightFill*** region).
- When unparsing, if the simple content region is larger than the encoded length of the element (as padded when dfdl:textPadKind is not 'none') then the remaining bytes, which are insufficient to hold another character code, are filled with dfdl:fillByte (Again, this is the grammar ***RightFill*** region.)

It is a Schema Definition Error if type is xs:string and dfdl:textPadKind is not 'none' and dfdl:lengthUnits is 'bytes' and dfdl:encoding is not an SBCS encoding and the XSD minLength facet is not zero. This prevents a scenario where validation against the XSD minLength facet is in characters, but padding would be performed in bytes.

12.3.7.2 Length of Simple Elements with Binary Representation

This section discusses the dfdl:lengthKind 'explicit' and 'prefixed' specified lengths for the different binary representations. When dfdl:lengthKind is 'implicit', see Section 12.3.3 dfdl:lengthKind 'implicit'.

The dfdl:lengthUnits can be 'bytes' or 'bits' unless otherwise stated. It is Schema Definition Error if dfdl:lengthUnits is 'characters'.

It is a Schema Definition Error if the specified dfdl:length for an element of dfdl:lengthKind 'explicit' is a string literal integer such that the length of the data exceeds the capacity of the simple type.

It is a Processing Error if the specified length for an element of dfdl:lengthKind 'prefixed' or 'explicit' (with dfdl:length an expression) is an integer such that the length of the data exceeds the capacity of the simple type.

12.3.7.2.1 Length of Base-2 Binary Number Elements

Non-floating point numbers with binary representation and dfdl:binaryNumberRep 'binary' are represented as a bit string which contains a base-2 representation.

The value of the specified length is constrained per the table below. The lengths are expressed in bits and are inclusive.

Type	Minimum value of length	Maximum value of length
xs:byte	2	8
xs:short	2	16
xs:int	2	32
xs:long	2	64
xs:unsignedByte	1	8
xs:unsignedShort	1	16
xs:unsignedInt	1	32
xs:unsignedLong	1	64
xs:nonNegativeInteger	1	Implementation-dependent (but not less than 64)
xs:integer	2	Implementation-dependent (but not less than 64)
xs:decimal	8[39]	Implementation-dependent (but not less than 64)

Table 23: Allowable Specified Lengths in Bits for Base-2 Binary Number Elements

See Section 13.7.1.1 Converting Base-2 Binary Numbers for details of the conversion to/from numeric values.

12.3.7.2.2 Length of Floating Point Binary Number Elements

For binary elements of types xs:float or xs:double, a specified length must be either exactly 4 bytes or exactly 8 bytes respectively.

The dfdl:lengthUnits property must be 'bytes'. It is a Schema Definition Error otherwise.

See Section 13.8 Properties Specific to Float/Double with Binary Representation.

12.3.7.2.3 Length of Packed Decimal Number Elements

Non-floating-point numbers with binary representation and dfdl:binaryNumberRep 'packed', 'bcd', or 'ibm4690Packed', are represented as a bit string of 4 bit nibbles. The term *packed decimal* is used to describe such numbers.

It is a Schema Definition Error if the specified length is not a multiple of 4 bits.

[39] Type decimal must be a minimum of 8 bits because lengthUnits 'bits' is not allowed for xs:decimal.

The maximum specified length of a packed decimal number is implementation-defined.

See Section 13.7 Properties Specific to Number with Binary Representation for details of the conversion of the packed decimal bit string to/from a numeric value.

12.3.7.2.4 Length of Binary Boolean Elements

The specified length of a binary element of type xs:boolean is as for type xs:unsignedInt described in Section 12.3.7.2.1 Length of Base-2 Binary Number Elements.

See also Section 13.10 Properties Specific to Boolean with Binary Representation for details of how the data is converted to/from a Boolean value.

12.3.7.2.5 Length of Base-2 Binary Calendar Elements

Calendars (types date, time, dateTime) with binary representation and dfdl:binaryCalendarRep 'binarySeconds' or 'binaryMilliseconds' are represented as a bit string which contains a base-2 representation. The specified length must be either exactly 4 bytes or exactly 8 bytes respectively.

The dfdl:lengthUnits property must be 'bytes'. It is a Schema Definition Error otherwise.

See Section 13.13 Properties Specific to Calendar with Binary Representation for details of how the data is converted to/from the calendar type.

12.3.7.2.6 Length of Packed Decimal Calendar Elements

Calendars (types date, time, dateTime) with binary representation and dfdl:binaryCalendarRep 'packed', 'bcd', or 'ibm4690Packed', are represented as a bit string of 4-bit nibbles. The term *packed decimal* is used to describe such calendars.

It is a Schema Definition Error if the specified length is not a multiple of 4 bits.

The maximum specified length of a packed decimal calendar is implementation-defined (but not less than 9 bytes, which corresponds to calendar pattern 'yyyyMMddhhmmssSSS')[40].

See Section 13.13 Properties Specific to Calendar with Binary Representation for details of how the data is converted to/from the calendar type.

12.3.7.2.7 Length of Binary Opaque Elements

The dfdl:lengthUnits property must be 'bytes'. It is a Schema Definition Error otherwise.

When unparsing a specified length element of type xs:hexBinary, and the simple content region is larger than the length of the element in the Infoset, then the remaining bytes are filled using the dfdl:fillByte property.

The dfdl:fillByte is **not** used to trim an element of type xs:hexBinary when parsing.

12.3.7.3 Length of Complex Elements

A complex element of specified length is defining a 'box' in which its child elements exist. An example of this would be a fixed-length record element with a variable number of children elements. The dfdl:lengthUnits may be 'bytes' or 'characters' and it is a Schema Definition Error otherwise.

It is possible that the children may not entirely fill the full length of the complex element. An example is a complex element with a specified length of 100 characters, which contains a sequence of child elements that use up less than 100 characters of data, perhaps because an optional element is not present. In this case the remaining unused data is called the ElementUnused region in the data syntax grammar of Section 9.2. Another example is a complex element with a specified length of 100 bytes, which contains a sequence of child elements the last of which has dfdl:lengthKind 'endOfParent', dfdl:representation 'text' and a multi-byte dfdl:encoding such that the element does not use up all the bytes of data. In this case the remaining unused bytes comprise the child element's ***RightFill*** region in the data syntax grammar of Section 9.2. In both examples, the unused area is skipped when parsing, and is filled with the dfdl:fillByte on unparsing.

Note that a poorly chosen value for dfdl:fillByte may fill the region with data that cannot be decoded in the character set encoding, resulting in a decode error when this data is subsequently parsed again. When dfdl:lengthUnits is 'characters' the value for dfdl:fillByte must be chosen to avoid this error.

[40] This is the smallest pattern that contains all the digit-only symbols. SSS is the minimum precision that must be supported for fractional seconds, but in can be more, hence why 'not less than 9 bytes'.

13 Simple Types

The dfdl:representation property identifies the physical representation of the element as text or binary. For some of the simple type and representation combinations there are additional properties that specify a further refinement of the representation.

These properties are described in relation to the logical type groupings of the simple types into Number, String, Calendar, Boolean, and Opaque groups, per Section 5.1 DFDL Simple Types.

13.1 Properties Common to All Simple Types

Property Name	Description
representation	Enum Valid values are dependent on logical type. **Number:** 'text', 'binary' **String:** representation is assumed to be 'text' and the dfdl:representation property is not examined **Calendar:** 'text', 'binary' **Boolean:** 'text', 'binary' **Opaque:** representation is assumed to be 'binary' and the dfdl:representation property is not examined. Annotation: dfdl:element, dfdl:simpleType

Table 24 Properties Common to All Simple Types

The permitted representation properties for each logical type are shown in Table 25: Logical Type to Representation properties

Logical type	dfdl:representation	Additional representation property
String	Assumed to be text	
Float, Double	text	**dfdl:textNumberRep:** standard
	binary	**dfdl:binaryFloatRep**: ieee, ibm390Hex
Decimal, Integer, nonNegativeInteger	text	**dfdl:textNumberRep:** standard, zoned
	binary	**dfdl:binaryNumberRep:** packed, bcd, ibm4690Packed, binary
Long, Int, Short, Byte, UnsignedLong, Unsignedint, Unsignedshort, UnsignedByte	text	**dfdl:textNumberRep:** standard, zoned
	binary	**dfdl:binaryNumberRep:** packed, bcd, ibm4690Packed, binary
DateTime, Date, Time	text	
	binary	**dfdl:binaryCalendarRep:** packed, bcd, ibm4690Packed, binarySeconds, binaryMilliseconds
Boolean	text	
	binary	
HexBinary	Assumed to be binary	

Table 25: Logical Type to Representation properties

13.2 Properties Common to All Simple Types with Text representation

Property Name	Description
textPadKind	Enum Valid values 'none', 'padChar'. Indicates whether to pad the data value on unparsing. This controls the contents of the ***LeftPadding*** and ***RightPadding*** regions of the data syntax grammar in Section 9.2 'none': No padding occurs. When dfdl:lengthKind is 'implicit' or 'explicit' (and dfdl:length is not an expression) the unparsed data value must match the expected length otherwise it is a Processing Error. 'padChar': The data value is padded using the dfdl:textStringPadCharacter, dfdl:textNumberPadCharacter, dfdl:textBooleanPadCharacter or dfdl:textCalendarPadCharacter depending on the type of the element. The padding characters populate the ***LeftPadding*** and/or ***RightPadding*** regions depending on dfdl:textStringJustification(see Section 13.4), dfdl:textNumberJustification (see Section 13.6), dfdl:textBooleanJustification (see Section 13.9), or dfdl:textCalendarJustification (see Section 13.12), depending on the type of the element. When dfdl:lengthKind is 'implicit' the data value is padded to the implicit length for the type. When dfdl:lengthKind is 'explicit' (and dfdl:length is not an expression) the data value is padded to the length given by the dfdl:length property. When dfdl:lengthKind is 'explicit' (and dfdl:length is an expression), 'delimited', 'prefixed', 'pattern' the data value is padded to the length given by the XSD minLength facet for type 'xs:string' or dfdl:textOutputMinLength property for other types. When dfdl:lengthKind is 'endOfParent' the data value is padded to the available length. Annotation: dfdl:element, dfdl:simpleType
textTrimKind	Enum Valid values 'none', 'padChar' Indicates whether to trim data on parsing. This controls the expected contents of the ***LeftPadding*** and ***RightPadding*** regions of the data syntax grammar in Section 9.2. When 'none' no trimming takes place. When 'padChar' the element is trimmed of the dfdl:textStringPadCharacter, dfdl:textNumberPadCharacter, dfdl:textBooleanPadCharacter or dfdl:textCalendarPadCharacter depending on the type of the element. The padding characters populate the ***LeftPadding*** and/or ***RightPadding*** regions depending on dfdl:textStringJustification, dfdl:textNumberJustification, or dfdl:textCalendarJustification, depending on the type of the element. Annotation: dfdl:element , dfdl:simpleType
textOutputMinLength	Non-negative Integer. Only used when dfdl:textPadKind is 'padChar' and dfdl:lengthKind is 'delimited', 'prefixed', 'pattern', 'explicit' (when dfdl:length is an expression) or 'endOfParent', and type is not xs:string Specifies the minimum content length during unparsing for simple types that do not allow the XSD minLength facet to be specified. For dfdl:lengthKind 'delimited', 'pattern' and 'endOfParent' the length units are always characters, for other dfdl:lengthKinds the length units are specified by the dfdl:lengthUnits property. If dfdl:textOutputMinLength is zero or less than the length of the representation text then no padding occurs. Annotation: dfdl:element, dfdl:simpleType

escapeSchemeRef	QName or empty String The name of the dfdl:defineEscapeScheme annotation that provides the additional properties used to describe the escape scheme. If the value is the empty string then escaping is explicitly turned off. See: Section 7.4 The dfdl:escapeScheme Annotation Element, and Section 7.3 The dfdl:defineEscapeScheme Defining Annotation Element. Annotation: dfdl:element, dfdl:simpleType

Table 26 Properties Common to All Simple Types with Text Representation

13.2.1 The dfdl:escapeScheme Properties

The dfdl:escapeScheme annotation is used within a dfdl:defineEscapeScheme annotation to group the properties of an escape scheme and allows a common set of properties to be defined that can be reused.

An escape scheme is needed when the content of a text element contains sequences of characters that are the same as an in-scope separator or terminator. If the characters are not escaped, a parser scanning for a separator or terminator would erroneously find the character sequence in the content.

An escape scheme defines the properties that describe the text escaping rules. There are two variants on such schemes:

- The use of a single escape character to cause the next character to be interpreted literally. The escape character itself is escaped by the escape-escape character.
- The use of a pair of escape strings to cause the enclosed group of characters to be interpreted literally. The ending escape string is escaped by the escape-escape character.

On parsing, the escape scheme is applied after pad characters are trimmed and on unparsing before pad characters are added. A pad character is not escaped by an escape character. When parsing, pad characters are trimmed without reference to an escape scheme. When unparsing, pad characters are added without reference to an escape scheme.

On unparsing, the application of escape scheme processing takes place before the application of the dfdl:emptyValueDelimiterPolicy property.

Property Name	Description
escapeKind	Enum Valid values 'escapeCharacter', 'escapeBlock' The type of escape mechanism defined in the escape scheme When 'escapeCharacter': On unparsing a single character of the data is escaped by adding a dfdl:escapeCharacter or dfdl:escapeEscapeCharacter immediately before it. The characters to escape are determined by property dfdl:escapeCharacterPolicy. On parsing any in-scope terminating delimiter encountered in the data is not interpreted as such when it is immediately preceded by the dfdl:escapeCharacter (when not itself preceded by the dfdl:escapeEscapeCharacter). Occurrences of the dfdl:escapeCharacter and dfdl:escapeEscapeCharacter are removed from the data as determined by property dfdl:escapeCharacterPolicy, unless the dfdl:escapeCharacter is preceded by the dfdl:escapeEscapeCharacter, or the dfdl:escapeEscapeCharacter does not precede the dfdl:escapeCharacter, respectively. It is a Schema Definition Error if the dfdl:escapeCharacter or dfdl:escapeEscapeCharacter is the same as the first character of an in-scope dfdl:separator or dfdl:terminator. When 'escapeBlock': On unparsing the entire data are escaped by adding dfdl:escapeBlockStart to the beginning and dfdl:escapeBlockEnd to the end of the data. The data is either always escaped or escaped when needed as specified by dfdl:generateEscapeBlock. If the data is escaped and contains the dfdl:escapeBlockEnd then first character of each appearance of the dfdl:escapeBlockEnd is escaped by the dfdl:escapeEscapeCharacter.

	On parsing the dfdl:escapeBlockStart string must be the first characters in the (trimmed) data in order to activate the escape scheme. The dfdl:escapeBlockStart string is removed from the beginning of the data. Until a matching dfdl:escapeBlockEnd string (that is, one not preceded by the dfdl:escapeEscapeCharacter) is found in the data, any in-scope terminating delimiter encountered in the data is not interpreted as such, and any dfdl:escapeEscapeCharacters are removed when they precede a dfdl:escapeBlockEnd string. The matching dfdl:escapeBlockEnd string is removed from the data.. The matching dfdl:escapeBlockEnd does not have to be the last character(s) in the (trimmed) data in order to de-activate the escape scheme. A dfdl:escapeBlockStart occurring anywhere in the data other than the first characters has no significance. Annotation: dfdl:escapeScheme
escapeCharacter	DFDL String Literal or DFDL Expression Specifies one character that escapes the subsequent character. Used when dfdl:escapeKind is 'escapeCharacter' It is a Schema Definition Error if dfdl:escapeCharacter is empty when dfdl:escapeKind is 'escapeCharacter' This property can be computed by way of an expression which returns a DFDL String Literal that represents a single character. The expression must not contain forward references to elements which have not yet been processed. *Escape and Quoting Character Restrictions*: The string literal is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is not allowed • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are not allowed It is a Schema Definition Error if the string literal contains any of the disallowed constructs. Escape characters contribute to the simple value region (SimpleLogicalValue or NilLiteralValue) of the field Annotation: dfdl:escapeScheme
escapeBlockStart	DFDL String Literal The string of characters that denotes the beginning of a sequence of characters escaped by a pair of escape strings. Used when dfdl:escapeKind is 'escapeBlock' It is a Schema Definition Error if dfdl:escapeBlockStart is empty when dfdl:escapeKind is 'escapeBlock' The string literal value is restricted in the same way as described in "Escape and Quoting Character Restrictions" in the description of the dfdl:escapeCharacter property. A dfdl:escapeBlockStart string contributes to the simple value region (SimpleLogicalValue or NilLiteralValue) of the field Annotation: dfdl:escapeScheme
escapeBlockEnd	DFDL String Literal The string of characters that denotes the end of a sequence of characters escaped by a pair of escape strings. Used when dfdl:escapeKind is 'escapeBlock' . It is a Schema Definition Error if dfdl:escapeBlockEnd is empty when dfdl:escapeKind is 'escapeBlock'. When parsing, it is a Processing Error if the end of the data for the element is reached and the escapeBlockEnd is not found in the data.

	The string literal value is restricted in the same way as described in "Escape and Quoting Character Restrictions" in the description of the escapeCharacter property. A dfdl:escapeBlockEnd string contributes to the simple value region (SimpleLogicalValue or NilLiteralValue) of the field Annotation: dfdl:escapeScheme
escapeEscapeCharacter	DFDL String Literal or DFDL Expression Specifies one character that escapes an immediately following dfdl:escapeCharacter or first character of dfdl:escapeBlockEnd. Used when dfdl:escapeKind is 'escapeCharacter' or 'escapeBlock'. This property can be computed by way of an expression which returns a DFDL String Literal that represents a single character. The expression must not contain forward references to elements which have not yet been processed. The string literal value is restricted in the same way as described in "Escape and Quoting Character Restrictions" in the description of the escapeCharacter property. If the empty string is specified then no escaping of escape characters occurs. It is explicitly allowed for both the dfdl:escapeCharacter and the dfdl:escapeEscapeCharacter to be the same character. In that case processing functions as if the dfdl:escapeCharacter escapes itself. Escape-escape characters contribute to the simple value region (SimpleLogicalValue or NilLiteralValue) of the field. Annotation: dfdl:escapeScheme
extraEscapedCharacters	List of DFDL String Literals A whitespace separated list of single characters that must be escaped in addition to the in-scope delimiters. If there are no extra characters to escape the property must be set to "". The string literal values are restricted in the same way as described in "Escape and Quoting Character Restrictions" in the description of the dfdl:escapeCharacter property. This property only applies on unparsing. Extra escaped characters contribute to the simple value region (SimpleLogicalValue or NilLiteralValue) of the field. Annotation: dfdl:escapeScheme
generateEscapeBlock	Enum Valid values 'always', 'whenNeeded' Controls when escaping is used on unparsing when dfdl:escapeKind is 'escapeBlock'. If 'always' then escaping is always occurs as described in dfdl:escapeKind. If 'whenNeeded' then escaping occurs as described in dfdl:escapeKind when the data contains any of the following: • any in-scope terminating delimiter • dfdl:escapeBlockStart at the start of the data • any dfdl:extraEscapedCharacters Annotation: dfdl:escapeScheme
escapeCharacterPolicy	Enum Valid values are ‘all’, ‘delimiters’. Controls when escape characters are removed during parsing, and output during unparsing, when dfdl:escapeKind is 'escapeCharacter'. When 'all':

	During unparsing the following are escaped as described in dfdl:escapeKind when they are in the data. • Any in-scope terminating delimiter by escaping its first character. • dfdl:escapeCharacter (escaped by dfdl:escapeEscapeCharacter) • any dfdl:extraEscapedCharacters During parsing, occurrences of dfdl:escapeCharacter and dfdl:escapeEscapeCharacter are interpreted and removed from the data as described in dfdl:escapeKind. When 'delimiters': During unparsing the following are escaped as described in dfdl:escapeKind when they are in the data. • Any in-scope terminating delimiter by escaping its first character. • dfdl:escapeCharacter (escaped by dfdl:escapeEscapeCharacter) During parsing, occurrences of dfdl:escapeCharacter and dfdl:escapeEscapeCharacter are interpreted and removed from the data as described in dfdl:escapeKind, except that dfdl:escapeCharacter is only removed when it immediately precedes an in-scope terminating delimiter. Annotation: dfdl:escapeScheme

Table 27 Escape Scheme Properties

13.2.1.1 Escape Scheme Example

Consider a dfdl:escapeScheme annotation with the following properties:

- dfdl:escapeBlockStart="start"
- dfdl:escapeBlockEnd="end"
- dfdl:escapeEscapeCharacter="#"

If this is used to serialize a DFDL Infoset element of type xs:string with value “A hash is a #”, then the value is wrapped with the dfdl:escapeBlockStart and dfdl:escapeBlockEnd, giving simple content "startA hash is a #end". If this data is parsed, the "#end" is treated as an escaped escape block end and the parse fails with a Processing Error, reporting that there is no escape block end in the data.

In this scenario, the data is not compliant with the escape scheme, and the DFDL unparser MUST issue a Processing Error.

Additional examples are in Appendix A: Escape Scheme Use Cases.

13.3 Properties for Bidirectional support for All Simple Types with Text representation

Bidirectional text is a feature expected in a future revision of the DFDL standard.

Property name	Description
textBidi	Enum Valid value is, 'no' This property exists in anticipation of future DFDL features that enable bidirectional text processing. Annotation: dfdl:element, dfdl:simpleType (representation text)

Table 28 Properties for Bidirectional support for All Simple Types with Text representation

13.4 Properties Specific to String

Property Name	Description
textStringJustification	Enum Valid values 'left', 'right', 'center' Unparsing:

	'left': Justifies to the left and adds padding chars to the string contents if the string is too short, to the length determined by the dfdl:textPadKind property. 'right': Justifies to the right and adds padding chars to the string contents if the string is too short, to the length determined by the dfdl:textPadKind property. 'center': Adds equal padding chars left and right of the string contents if the string is too short, to the length determined by the dfdl:textPadKind property. It adds one extra padding char on the left if needed. Parsing: 'left': Trims any pad characters from the right of the string, according to dfdl:textTrimKind property. 'right': Trims any pad characters from the left of the string, according to dfdl:textTrimKind property. 'center' Trims any pad characters from the left and right of the string, according to dfdl:textTrimKind property. Annotation: dfdl:element, dfdl:simpleType
textStringPadCharacter	DFDL String Literal The value that is used when padding or trimming string elements. The value can be a single character or a single byte. If a character, then it can be specified using a literal character or using DFDL entities. If a byte, then it must be specified using a single byte value entity otherwise it is a Schema Definition Error If a pad character is specified when dfdl:lengthUnits is 'bytes' then the pad character must be a single-byte character. If a pad byte is specified when dfdl:lengthUnits is 'characters' then • the encoding must be a fixed-width encoding • padding and trimming must be applied using a sequence of N pad bytes, where N is the width of a character in the fixed-width encoding. *Padding Character Restrictions:* The string literal is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is allowed. • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are not allowed It is a Schema Definition Error if the string literal contains any of the disallowed syntax. Annotation: dfdl:element, dfdl:simpleType
truncateSpecifiedLengthString	Enum Valid values are 'yes', 'no' Used on unparsing only. 'yes' means if the logical type is xs:string and the value is longer than the specified length, the string is truncated to this length. (See Section 12.3.7 Elements of Specified Length.) No Processing Error is raised. This property is needed when a DFDL schema has specified lengths for strings. The strings in an Infoset being unparsed do not necessarily fit within those specified lengths. This property provides the means to express whether this is an error, or the strings can be truncated to fit. The position from which data is truncated is determined by the value of the dfdl:textStringJustification property. If the value of the

	dfdl:textStringJustification property is 'left', data is truncated from the right; if the value of the dfdl:textStringJustification property is 'right', data is truncated from the left. However, if the value of the dfdl:textStringJustification property is 'center', truncation does not occur, and a Processing Error occurs if the value is too long. When unparsing, Validation Errors cannot be prevented by truncation as validation takes place on the augmented Infoset, before any truncation has occurred. Annotation: dfdl:element, dfdl:simpleType

Table 29 Properties Specific to String

13.5 Properties Specific to Number with Text or Binary Representation

Property Name	Description
decimalSigned	Enum Valid values are 'yes', 'no' Indicates whether an xs:decimal element is signed. See 13.6.2 Converting logical numbers to/from text representation and 13.7.1 Converting Logical Numbers to/from Binary to see how this affects the presence of the sign in the data stream. 'yes' means that the xs:decimal element is signed 'no' means that the xs:decimal element is not signed Annotation: dfdl:element, dfdl:simpleType

Table 30 Properties Specific to Number with Text or Binary Representation

13.6 Properties Specific to Number with Text Representation

There are many properties for describing textual number representations. The properties deal with the representation of the numeric value only. Other symbols adjacent to the textual representation of a number, such as currency symbols, percent signs, or coordinate axis indicators, are not considered part of the value representation.

Property Name	Description
textNumberRep	Enum Valid values are 'standard', 'zoned' 'standard' means represented as characters in the character set encoding specified by the dfdl:encoding property. 'zoned' means represented as a zoned decimal in the character set encoding specified by the dfdl:encoding property. In zoned representation each decimal digit is stored in one character code point (usually 1 byte), with the least-significant four bits encoding the digit value 0 through 9. The most-significant four bits, called the "zone" bits, are usually set to a fixed value Typically these zone bits are hex F in EBCDIC encodings or 3 in ASCII encodings so that the byte holds a character value corresponding to the digit. However, in the first or last character code the zone bits are modified to represent the sign of the number. This is called *overpunched sign* since zoned representation originated when computers used punched cards for data. Which characters are used to represent modified ('overpunched') positive and negative signs varies by encoding, COBOL compiler, and system. The code points are fixed for EBCDIC systems but not for ASCII. In EBCDIC-based encodings, code points 0xC0 to 0xC9 or 0xF0 to 0xF9 represent a positive sign and digits 0 to 9 (these byte ranges correspond typically to characters '{ABCDEFGHI' or '0123456789'), and code points 0xD0 to 0xD9 or 0xB0 to 0xB9 represent a negative sign and digits 0 to 9

	(these byte ranges correspond typically to characters '}JKLMNOPQR' or '^£¥·©§¶¼½¾ '). On parsing both ranges are accepted. On unparsing the range 0xC0 to 0xC9 are produced for positive signs and the range 0xD0 to 0xD9 are produced for negative signs. For ASCII-based encodings see the property dfdl:textZonedSignStyle. Zoned is not supported for float and double numbers. Base 10 is assumed, and the encoding must be for an EBCDIC or ASCII compatible encoding. It is a Schema Definition Error if any of these requirements are not met. Annotation: dfdl:element, dfdl:simpleType
textNumberJustification	Enum Valid values 'left', 'right', 'center' Controls how the data is padded or trimmed on parsing and unparsing. Behavior as for dfdl:textStringJustification. Annotation: dfdl:element, dfdl:simpleType
textNumberPadCharacter	DFDL String Literal The value that is used when padding or trimming number elements. The value can be a single character or a single byte. If a character, then it can be specified using a literal character or using DFDL entities. If a byte, then it must be specified using a single byte value entity If a pad character is specified when dfdl:lengthUnits is 'bytes' then the pad character must be a single-byte character. If a pad byte is specified when dfdl:lengthUnits is 'characters' then • the encoding must be a fixed-width encoding • padding and trimming must be applied using a sequence of N pad bytes, where N is the width of a character in the fixed-width encoding. When parsing, if the pad character is '0' and dfdl:textTrimKind is 'padChar' then the SimpleContent region is trimmed of the '0' characters as defined by the trimming rules. If at least one '0' character is removed and the trimmed text causes a Processing Error when parsed, a single '0' character is re-instated, and the text is parsed again. This is to handle the case when '0' characters are trimmed away leaving no digits. This rule also applies when the pad character is a DFDL character entity equivalent to '0'. This rule does not apply when the pad character is any other character nor when a pad byte is specified. The string literal value is restricted in the same way as described in "Pad Character Restrictions" in the description of the dfdl:textStringPadCharacter property. Annotation: dfdl:element, dfdl:simpleType
textNumberPattern	String Defines the ICU-like pattern that describes the format of the text number. The pattern defines where grouping separators, decimal separators, implied decimal points, exponents, positive signs and negative signs appear. It permits definition by either digits/fractions or significant digits. Allows rounding. When dfdl:textNumberRep is 'standard' this property only applies when dfdl:textStandardBase is 10. When dfdl:textNumberRep is 'standard' and dfdl:textStandardBase is not 10 the number is represented as the minimum number of characters to represent the digits. There is no sign or virtual decimal point. The syntax of dfdl:textNumberPattern is described in Section 13.6.1 The dfdl:textNumberPattern Property

	Annotation: dfdl:element, dfdl:simpleType
textNumberRounding	Enum Specifies how rounding is controlled during unparsing. Valid values 'pattern', 'explicit' When dfdl:textNumberRep is 'standard' this property only applies when dfdl:textStandardBase is 10. If 'pattern' then rounding takes place according to the pattern. A rounding increment may be specified in the dfdl:textNumberPattern using digits '1' though '9', otherwise rounding is to the width of the pattern. The rounding mode is always 'roundHalfEven'. If 'explicit' then the rounding increment is specified by the dfdl:textNumberRoundingIncrement property, and any digits '1' through '9' in the dfdl:textNumberPattern are treated as digit '0'. The rounding mode is specified by the dfdl:textRoundingMode property. To disable rounding, use 'explicit' in conjunction with 'roundUnnecessary' for the dfdl:textNumberRoundingMode. If rounding is disabled, then any need for rounding is treated as a Processing Error. Annotation: dfdl:element, dfdl:simpleType
textNumberRoundingMode	Enum Specifies how rounding occurs during unparsing, when dfdl:textNumberRounding is 'explicit'. When dfdl:textNumberRep is 'standard' this property only applies when dfdl:textStandardBase is 10. To switch off rounding, use 'roundUnnecessary'. Valid values 'roundCeiling', 'roundFloor', 'roundDown', 'roundUp', 'roundHalfEven', 'roundHalfDown', 'roundHalfUp', 'roundUnnecessary' The enum values have these rounding directions: • 'roundCeiling' - toward positive infinity. • 'roundFloor' - toward negative infinity • 'roundDown' - toward zero • 'roundUp' - away from zero • 'roundHalfEven' - toward nearest neighbor, except when both neighbors are equidistant, in which case round towards the even neighbor. • 'roundHalfDown' - toward nearest neighbor, except when both neighbors are equidistant, in which case round down. • 'roundHalfUp' - toward nearest neighbor, except when both neighbors are equidistant, in which case round up. • 'roundUnnecessary' - no rounding. If rounding is necessary it is a Processing Error. Annotation: dfdl:element, dfdl:simpleType
textNumberRoundingIncrement	Double Specifies the rounding increment to use during unparsing, when dfdl:textNumberRounding is 'explicit'. When dfdl:textNumberRep is 'standard' this property only applies when dfdl:textStandardBase is 10. A negative value is a Schema Definition Error. Annotation: dfdl:element, dfdl:simpleType
textNumberCheckPolicy	Enum Values are 'strict' and 'lax'. Indicates how lenient to be when parsing against the dfdl:textNumberPattern.

	When dfdl:textNumberRep is 'standard' this property only applies when dfdl:textStandardBase is 10. If 'lax' and dfdl:textNumberRep is 'standard' then behavior is implementation-defined, but typically includes grouping separators are ignored, leading and trailing whitespace is ignored, leading zeros are ignored, and quoted characters may be omitted. If 'lax' and dfdl:textNumberRep is 'zoned' then positive punched data is accepted when parsing an unsigned type, and unpunched data is accepted when parsing a signed type If 'strict' and dfdl:textNumberRep is 'standard' then the data must follow the pattern with the exceptions that digits 0-9, decimal separator and exponent separator are always recognized and parsed. If 'strict' and dfdl:textNumberRep is 'zoned' then the data must follow the pattern. On unparsing the pattern is always followed and follow the rules in 13.6.2 Converting logical numbers to/from text representation. Annotation: dfdl:element, dfdl:simpleType
textStandardDecimalSeparator	List of DFDL String Literals or DFDL Expression The decimal separator is the punctuation mark which separates the integer part of a decimal or floating point number from the fractional part. It is usually a period or comma depending on locale of the data. This property defines a whitespace separated list of single characters that appear (individually) in the data as the decimal separator. This property is applicable, when dfdl:textNumberRep is 'standard' and dfdl:textStandardBase is 10. It must be set if dfdl:textNumberPattern contains a decimal separator symbol ("."), or the E or @ symbols. (it is a Schema Definition Error otherwise.) Empty string is not an allowable value. This property can be computed by way of an expression which returns a DFDL String Literal that represents a single character. The expression must not contain forward references to elements which have not yet been processed. *Text Number Character Restrictions:* The string literal is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is not allowed. • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are not allowed It is a Schema Definition Error if the string literal contains any of the disallowed syntax constructs. In addition, it is a Schema Definition Error if any of the string literal values for this property are digits 0-9. Annotation: dfdl:element, dfdl:simpleType
textStandardGroupingSeparator	DFDL String Literal or DFDL Expression The grouping separator is the punctuation mark which separates the clusters of integer digits to improve readability. This property defines the single character that can appear in the data as the grouping separator. This property is applicable when dfdl:textNumberRep is 'standard' and dfdl:textStandardBase is 10. It must be set if dfdl:textNumberPattern contains a grouping separator symbol (it is a Schema Definition Error otherwise.) Empty string is not an allowable value. This property can be computed by way of an expression which returns a DFDL String Literal that represents a single character. The expression

	must not contain forward references to elements which have not yet been processed. The string literal value is restricted in the same way as described in "Text Number Character Restrictions" in the description of the dfdl:textStandardDecimalSeparator property. See also Section 13.6.1.1 dfdl:textNumberPattern for dfdl:textNumberRep 'standard' for additional details about grouping separators. Annotation: dfdl:element, dfdl:simpleType
textStandardExponentRep	DFDL String Literal or DFDL Expression Defines the actual character(s) that appear in the data as the exponent indicator. If the empty string is specified then no exponent character is used. This property is applicable when dfdl:textNumberRep is 'standard' and dfdl:textStandardBase is 10. Empty string is an allowable value, so that formats like NNN+M (meaning NNN x 10 with MM exponent) can be expressed. This property must be set even if the dfdl:textNumberPattern does not contain an 'E' (exponent) character. It is a Schema Definition Error if this property is not set or in scope for any number with dfdl:representation 'text'. This property can be computed by way of an expression which returns a DFDL String Literal. The expression must not contain forward references to elements which have not yet been processed. The string literal value is restricted in the same way as described in "Text Number Character Restrictions" in the description of the dfdl:textStandardDecimalSeparator property. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. Annotation: dfdl:element, dfdl:simpleType
textStandardInfinityRep	DFDL String Literal The value used to represent infinity. Infinity is represented as a string with the positive or negative prefixes and suffixes from the dfdl:textNumberPattern applied. This property is applicable when dfdl:textNumberRep is 'standard', dfdl:textStandardBase is 10 and the simple type is float or double. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. The string literal value is restricted in the same way as described in "Text Number Character Restrictions" in the description of the dfdl:textStandardDecimalSeparator property. It is a Schema Definition Error if empty string found as the property value. Annotation: dfdl:element, dfdl:simpleType
textStandardNaNRep	DFDL String Literal The value used to represent NaN. NaN is represented as a string and the positive or negative prefixes and suffixes from the dfdl:textNumberPattern are not used. This property is applicable when dfdl:textNumberRep is 'standard', dfdl:textStandardBase is 10 and the simple type is float or double. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser.

	The string literal value is restricted in the same way as described in "Text Number Character Restrictions" in the description of the dfdl:textStandardDecimalSeparator property. It is a Schema Definition Error if empty string found as the property value. Annotation: dfdl:element, dfdl:simpleType
textStandardZeroRep	List of DFDL String Literals Valid values: empty string, any character string The whitespace separated list of alternative DFDL String Literals that are equivalent to zero, for example the characters 'zero'. The representation is examined for a match to one of the values of this property after padding has been trimmed away. On unparsing the first value is used. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. The empty string means that there is no special literal string for zero. This property is applicable when dfdl:textNumberRep is 'standard' and dfdl:textStandardBase is 10. Each string literal in the list is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed. • DFDL Byte Value entities (%#rXX;) are not allowed. • DFDL Character class entities NL and ES are not allowed. • DFDL Character class entities WSP, WSP+, and WSP* are allowed. However, the WSP* entity cannot appear on its own as one of the string literals in the list. It must be used in combination with other text characters or entities so as to describe a representation that cannot ever be an empty string. It is a Schema Definition Error if the string literal contains any of the disallowed syntax constructs. Annotation: dfdl:element, dfdl:simpleType
textStandardBase	Non-negative Integer Valid Values 2, 8, 10, 16 Indicates the number base. Only used when dfdl:textNumberRep is 'standard'. When base is not 10, xs:decimal, xs:float, and xs:double are not supported. When dfdl:textNumberRep is 'zoned' dfdl:textStandardBase is not used and base 10 is assumed. Annotation: dfdl:element, dfdl:simpleType
textZonedSignStyle	Enum Specifies the code points that are used to modify the sign nibble of the byte containing the sign, when the dfdl:encoding is an ASCII-derived character set encoding. The location of this sign nibble is indicated in the dfdl:textNumberPattern. This property is applicable when dfdl:textNumberRep is 'zoned'. Used only when dfdl:encoding is an ASCII-derived character set encoding. The encoding must provide the character to single byte code point mapping used by the specified value of dfdl:textZonedSignStyle, as stated below.

	Valid values 'asciiStandard', 'asciiTranslatedEBCDIC', 'asciiCARealiaModified', and 'asciiTandemModified' Which characters are used to represent modified (also called 'overpunched') positive and negative signs, varies by encoding, COBOL compiler, and system. The code points are fixed for EBCDIC systems but not for ASCII. In ASCII-based encodings, this property is used to determine how signs are expressed for zoned numbers. • asciiStandard: ASCII characters '0123456789' represent a positive sign and the corresponding digit. (Sign nibble for '+' is 0x3, which is the high nibble of these code points unmodified.) ASCII characters 'pqrstuvwxy' represent negative sign and digits 0 to 9. (Code points 0x70 to 0x79) • asciiTranslatedEBCDIC: The overpunched character is the ASCII equivalent of the typical EBCDIC above. So, the characters '{ABCDEFGHI' still represent a positive sign and digits 0 to 9. (These are code points 0x7B, 0x41 through 0x49). The characters '}JKLMNOPQR' still represent negative sign and digits 0 to 9. (These are code points 0x7D, 0x4A through 0x52). This case comes up if EBCDIC zoned decimal data is translated to ASCII as if it were textual data.) • asciiCARealiaModified41: In this style, the ASCII characters '0123456789' represent positive sign and digits 0 to 9 as in asciiStandard. However, ASCII characters from code points 0x20 to 0x29 are used for negative sign and the corresponding decimal digit. This doesn't translate well into printing characters. These characters include the space (' ') for zero, characters '!"#$%&' for 1 through 6, the single quote character "'" for 7, and the parenthesis '()' for 8 and 9. • asciiTandemModified: In this style the ASCII characters '0123456789' represent positive sign and digits 0 to 9, but code points 0x80 to 0x89 are used to represent negative sign and a digit. There are no corresponding code points in the standard ASCII encoding since these values are all above 128 (decimal). This means the resultant bytes are not code points in standard ASCII, so the schema must specify an encoding like ISO-8859-1 for such zoned decimals to parse without an encoding error. (Note that neither ISO-8859-1 encoding, nor Unicode have assigned glyphs for these code points. They are considered control characters.) Annotation: dfdl:element, dfdl:simpleType

Table 31 Properties Specific to Number with Text Representation

The dfdl:textStandardDecimalSeparator, dfdl:textStandardGroupingSeparator, dfdl:textStandardExponentRep, dfdl:textStandardInfinityRep, dfdl:textStandardNaNRep, and dfdl:textStandardZeroRep must all be distinct, and it is a Schema Definition Error otherwise. Note that if dfdl:textStandardDecimalSeparator, dfdl:textStandardGroupingSeparator, or dfdl:textStandardExponentRep are expressions, this checking can only be carried out during processing (parsing or unparsing.)

Implementation note: This rule is in the interests of clarity and is an extra constraint compared to ICU.

[41] Reference for this CA Realia 0x20 overpunch for negative sign is the article: "EBCDIC to ASCII Conversion of Signed Fields" [CARealia] where it says:

COBOL compilers that run on ASCII platforms have a "signed" data type that operates in a similar manner to the EBCDIC Signed field -- that is, they over punch the sign on the LSD (Least Significant Digit). However, this is not standardized in ASCII, and different compilers use different overpunch codes. For example, Computer Associates' Realia compiler uses a 30 hex for positive values and a 20 hex for negative values, but Micro Focus® and Microsoft® use 30 hex for positive values and 70 hex for negative values.

13.6.1 The dfdl:textNumberPattern Property

The dfdl:textNumberPattern describes how to parse and unparse text representations of number logical types with base 10.

The length of the representation of the number is determined first, and the number pattern is used only for conversion of the content text to and from a numeric logical Infoset value.

The pattern described below is derived from the ICU DecimalFormat class described here: [ICUDecimal]

The pattern is an ICU-like syntax that defines where grouping separators, decimal separators, implied decimal points, exponents, positive signs and negative signs appear. It permits definition by either digits/fractions or significant digits.

13.6.1.1 dfdl:textNumberPattern for dfdl:textNumberRep 'standard'

When dfdl:textNumberRep is 'standard' this property only applies when dfdl:textStandardBase is 10.

The pattern comes in two parts separated by a semi-colon. The first is mandatory and applies to positive numbers, the second is optional and applies to negative numbers.

Examples: The first shows digits/fractions and positive/negative signs, the second shows exponent, the third shows virtual decimal point, the fourth shows scaling position.

```
+###,##0.00;(###,##0.00)

##0.0#E0

000V00

PPP0000
```

The 'V' symbol is used to indicate the location of an implied decimal point for fixed point number representations. (This is an extension to the ICU pattern language.)

The 'P' symbol is used to indicate that a decimal scaling factor needs to be applied. (This is an extension to the ICU pattern language.)

The actual grouping separator, decimal separator and exponent characters are defined independently of the pattern.

The actual positive sign and negative sign are defined within the pattern itself.

Many characters in a pattern are taken literally; they are matched during parsing and output unchanged during unparsing. Special characters, on the other hand, stand for other characters, strings, or classes of characters. For example, the '#' character is replaced by a digit.

To insert a special character in a pattern as a literal, that is, without any special meaning, the character, or a string containing it must be surrounded by quote symbols. There are some exceptions to this which are noted below.

Symbol	Location	Meaning
0	Number	Digit
1-9	Number	'1' through '9' indicates rounding.
#	Number	Digit, zero shows as absent
.	Number	Decimal separator or monetary decimal separator
-	Number	Minus sign
,	Number	Grouping separator
E	Number	Separates mantissa and exponent in scientific notation. Need not be quoted in prefix or suffix if use is unambiguous.
+	Exponent	Prefix positive exponents with plus sign. Need not be quoted in prefix or suffix if use is unambiguous.
;	Subpattern boundary	Separates positive and negative subpatterns

'	Prefix or suffix	Used to escape special characters in a prefix or suffix, for example, `'#'#` formats 123 to `#123`. To create a single quote itself, use two in a row: `#o''clock`. Multiple special characters may be escaped either by quoting individually or by quoting a containing string, for example `'VALUE' #` and `'V'ALU'E' #` both format 123 to `VALUE 123`.
*	Prefix or suffix boundary	Pad escape, precedes pad character
V	Number	Virtual decimal point marker. Only used with decimal, float and double simple types.
P	Number	Decimal scaling position. Only used with decimal, float and double simple types.
@	Number	Significant digits specifier. Only used with decimal simple type. Controls number of significant digits when used alone or in conjunction with the # character.

Table 32 dfdl:textNumberPattern Special Characters

A pattern contains a positive and negative subpattern, for example, "#,##0.00;(#,##0.00)". Each subpattern has a prefix, a numeric part, and a suffix. If there is no explicit negative subpattern, the negative subpattern is the minus sign prefixed to the positive subpattern. That is, "0.00" alone is equivalent to "0.00;-0.00". If there is an explicit negative subpattern, it serves only to specify the negative prefix and suffix; the number of digits, minimal digits, and other characteristics are ignored in the negative subpattern. That means that "#,##0.0#;(#)" has precisely the same result as "#,##0.0#;(#,##0.0#)".

The prefixes, suffixes, and various symbols used for infinity, digits, grouping separators, decimal separators, etc. may be set to arbitrary values, and they appear properly during unparsing. However, care must be taken that the symbols and strings do not conflict, or parsing will be unreliable. For example, either the positive and negative prefixes or the suffixes must be distinct for parse to be able to distinguish positive from negative values.

The *grouping separator* is a character that separates clusters of integer digits to make large numbers more legible. It commonly used for thousands, but in some locales it separates ten-thousands. The *grouping size* is the number of digits between the grouping separators, such as 3 for "100,000,000" or 4 for "1 0000 0000". There are two different grouping sizes: One used for the least significant integer digits, the *primary grouping size*, and one used for all others, the *secondary grouping size*. In most locales these are the same, but sometimes they are different. For example, if the primary grouping interval is 3, and the secondary is 2, then this corresponds to the pattern "#,##,##0", and the number 123456789 is formatted as "12,34,56,789". If a pattern contains multiple grouping separators, the interval between the last one and the end of the integer defines the primary grouping size, and the interval between the last two defines the secondary grouping size. All others are ignored, so "#,##,###,####" == "###,###,####" == "##,#,###,####".

The P symbol is used to derive the location of an assumed decimal point when the point is not within the number that appears in the data. It acts as a decimal scaling factor.

The symbol P can be specified only as a continuous string of Ps in the leftmost or rightmost digit positions in the vpinteger region of the pattern.

It is a Schema Definition Error if any symbols other than "0", "1" through "9" or # are used in the vpinteger region of the pattern.

Examples

Data Representation	Pattern	Value
123	PP000	0.00123
123	000PP	12300

Table 33 Examples of P Symbol in the dfdl:textNumberPattern Property

```
pattern    := subpattern (';' subpattern)?
subpattern := prefix? ((number exponent?)| vpinteger) suffix?
number     := (integer ('.' fraction)?) | sigdigits
vpinteger := pinteger | (vinteger exponent?)
```

```
pinteger := ('P' 'P'* '0'* '0') | (integer 'P'* 'P' )
vinteger := ('#'* '0'* 'V' '0'* '0') | (integer 'V')
prefix     := '\u0000'..'\uFFFD' - specialCharacters
suffix     := '\u0000'..'\uFFFD' - specialCharacters
integer    := '#'* '0'* '0'
fraction   := '0'* '#'*
sigDigits  := '#'* '@' '@'* '#'*
exponent   := 'E' '+'? '0'* '0'
padSpec    := '*' padChar
padChar    := '\u0000'..'\uFFFD' - quote

Notation:
  X*       0 or more instances of X
  X?       0 or 1 instances of X
  X|Y      either X or Y
  C..D     any character from C up to D, inclusive
  S-T      characters in S, except those in T
```

Figure 4 dfdl:textNumberPattern BNF syntax

The first subpattern is for positive numbers. The second (optional) subpattern is for negative numbers.

Not indicated in the BNF syntax above:

- The grouping separator ',' can occur inside the integer region, between any two pattern characters of that region, as long as the number region is not followed by an exponent region.
- Two grouping intervals are recognized: That between the decimal point and the first grouping symbol, and that between the first and second grouping symbols. These intervals are identical in most locales, but in some locales they differ. For example, the pattern "#,##,###" formats the number 123456789 as "12,34,56,789".
- The pad specifier padSpec may appear before the prefix, after the prefix, before the suffix, after the suffix, or not at all.
- In place of '0', the digits '1' through '9' in the number or vpinteger region may be used to indicate a rounding increment.

The term *maximum fraction digits* is the total number of '0' and '#' characters in the fraction sub-pattern above.

The term *minimum fraction digits* is the total number of '0' characters (only) in the fraction sub-pattern above.

The term *maximum integer digits* is a limit that is implementation-dependent but MUST be at least 20 (which is the number of digits in a base 10 unsigned long).[42]

The term *minimum integer digits* is the total number of '0' characters (only) in the integer sub-pattern above.

Parsing

During parsing, grouping separators are removed from the data.

Unparsing

Unparsing is guided by several parameters all of which can be specified using a pattern. The following description applies to formats that do not use scientific notation.

If the number of actual integer digits exceeds the *maximum integer digits*, then only the least significant digits are output. For example, 1997 is formatted as "97" if the maximum integer digits are 2.

If the number of actual integer digits is less than the *minimum integer digits*, then leading zeros are added. For example, 1997 is formatted as "01997" if the minimum integer digits are 5.

If the number of actual fraction digits exceeds the *maximum fraction digits*, then half-even rounding is performed to the maximum fraction digits. For example, 0.125 is formatted as "0.12" if the maximum fraction digits are 2. This behavior can be changed by specifying a rounding increment and a rounding mode.

If the number of actual fraction digits is less than the *minimum fraction digits*, then trailing zeros are added. For example, 0.125 is formatted as "0.1250" if the minimum fraction digits are 4.

Trailing fractional zeros are not output if they occur *j* positions after the decimal, where *j* is less than the maximum fraction digits. For example, 0.10004 is formatted as "0.1" if the maximum fraction digits are four or less.

Special Values

[42] Implementations which use current versions of the popular ICU library will allow 309 digits as *maximum integer digits*.

NaN is represented as a string determined by the dfdl:textStandardNaNRep property. This is the only value for which the prefixes and suffixes are not used.

Infinity is represented as a string with the positive or negative prefixes and suffixes applied. The infinity string is determined by the dfdl:textStandardInfinityRep property.

Scientific Notation

Numbers in scientific notation are expressed as the product of a mantissa and a power of ten, for example, 1234 can be expressed as 1.234 x 10^3. The mantissa is typically in the half-open interval [1.0, 10.0) or sometimes [0.0, 1.0), but it need not be. In a pattern, the exponent character immediately followed by one or more digit characters indicates scientific notation. Example: "0.###E0" formats the number 1234 as "1.234E3".

The number of digit characters after the exponent character gives the minimum exponent digit count. There is no maximum. Negative exponents are formatted using the minus sign, *not* the prefix and suffix from the pattern. This allows patterns such as "0.###E0 m/s". To prefix positive exponents with a plus sign, specify '+' between the exponent and the digits: "0.###E+0" produces data like "1E+1", "1E+0", "1E-1", etc.

The minimum number of integer digits is achieved by adjusting the exponent. Example: 0.00123 formatted with "00.###E0" yields "12.3E-4". This only happens if there is no maximum number of integer digits. If there is a maximum, then the minimum number of integer digits is fixed at one.

The maximum number of integer digits, if present, specifies the exponent grouping. The most common use of this is to generate *engineering notation*, in which the exponent is a multiple of three, e.g., "##0.###E0". The number 12345 is formatted using "##0.####E0" as "12.345E3".

When using scientific notation, the formatter controls the digit counts using significant digits logic. The maximum number of significant digits limits the total number of integer and fraction digits that are shown in the mantissa; it does not affect parsing. For example, 12345 formatted with "##0.##E0" is "12.3E3". .

Exponential patterns must not contain grouping separators.

Significant Digits

The '@' pattern character can be used with the '#' to control how many integer and fraction digits are needed to display the specified number of significant digits. The '@' only affects unparsing behavior. Examples:

Pattern	Minimum significant digits	Maximum significant digits	Number	Formatted Output
@@@	3	3	12345	12300
@@@	3	3	0.12345	0.123
@@##	2	4	3.14159	3.142
@@##	2	4	1.23004	1.23

Table 34 Significant Digits '@' Symbol in the dfdl:textNumberPattern Property

Significant digit counts may be expressed using patterns that specify a minimum and maximum number of significant digits. These are indicated by the '@' and '#' characters. The minimum number of significant digits is the number of '@' characters. The maximum number of significant digits is the number of '@' characters plus the number of '#' characters following on the right. For example, the pattern "@@@" indicates exactly 3 significant digits. The pattern "@##" indicates from 1 to 3 significant digits. Trailing zero digits to the right of the decimal separator are suppressed after the minimum number of significant digits have been shown. For example, the pattern "@##" formats the number 0.1203 as "0.12".

If a pattern uses significant digits, it must not contain a decimal separator, nor the '0' pattern character. Patterns such as "@00" or "@.###" are disallowed.

Any number of '#' characters may be prepended to the left of the leftmost '@' character. These have no effect on the minimum and maximum significant digits counts but may be used to position grouping separators. For example, "#,#@#" indicates a minimum of one significant digit, a maximum of two significant digits, and a grouping size of three.

The number of significant digits has no effect on parsing.

Significant digits may be used together with exponential notation. For example, the pattern "@@###E0" is equivalent to "0.0###E0".

The '@' pattern character can be used only in 'standard' textNumberRep (not 'zoned') and excludes the 'P' and 'V' pattern characters. It is a Schema Definition Error if the '@' pattern character appears in 'zoned' textNumberRep, or in conjunction with the 'P' or 'V' pattern characters.

Padding

Padding may be specified through the pattern syntax. In a pattern the pad escape character, followed by a single pad character, causes padding to be parsed and formatted. The pad escape character is '*'. For example, "*x#,##0.00" formats 123 to "xx123.00", and 1234 to "1,234.00".

When padding is in effect, the width of the positive subpattern, including prefix and suffix, determines the format width. For example, in the pattern "* #0 o"clock", the format width is 10.

The width is counted in 16-bit code units.

Some parameters which usually do not matter have meaning when padding is used, because the pattern width is significant with padding. In the pattern "* ##,##,#,##0.##", the format width is 14. The initial characters "##,##," do not affect the grouping size or maximum integer digits, but they do affect the format width.

Padding may be inserted at one of four locations: before the prefix, after the prefix, before the suffix, or after the suffix. If there is no prefix, before the prefix and after the prefix are equivalent, likewise for the suffix.

When specified in a pattern, the 32-bit codepoint immediately following the pad escape is the pad character. This may be any character, including a special pattern character. That is, the pad escape *escapes* the following character. If there is no character after the pad escape, then the pattern is illegal.

Note: Padding specified through the pattern syntax is distinct from, and in addition to, padding specified using dfdl:textPadKind.

Rounding

How rounding is controlled is given by dfdl:textNumberRounding. The rounding increment may be specified in the dfdl:textNumberPattern itself using digits '1' through '9' or using an explicit increment in dfdl:textNumberRoundingIncrement. For example, 1230 rounded to the nearest 50 is 1250. 1.234 rounded to the nearest 0.65 is 1.3.

- Rounding only affects the string produced by unparsing. It does not affect parsing or change any numerical values.
- In a pattern, digits '1' through '9' specify rounding, but otherwise behave identically to digit '0'. For example, "#,#50" specifies a rounding increment of 50.
- Using digits in a pattern, rounding is always 'half even', meaning rounds towards the nearest integer, or towards the nearest even integer if equidistant.

Using an explicit rounding increment, dfdl:textNumberRoundingMode determines how values are rounded.

13.6.1.2 dfdl:textNumberPattern for dfdl:textNumberRep 'zoned'

When dfdl:textNumberRep is 'zoned' a subset of the number pattern language described in Section 13.6.1.1 dfdl:textNumberPattern for dfdl:textNumberRep 'standard' is used.

Only the pattern for positive numbers is used. It is a Schema Definition Error if the negative pattern is specified.

In addition, only the following pattern characters may be used:

- '+' must be present at the beginning or end of the pattern to indicate whether the leading or trailing digit carries the overpunched sign, if the logical type is signed
- '+' may be present at the beginning or end of the pattern to indicate whether the leading or trailing digit carries the overpunched sign, if the logical type is unsigned. If logical type is unsigned and dfdl:textNumberPolicy 'lax' specified it is a Schema Definition Error if no '+' is present.
- 'V' may be used to indicate the location of an implied decimal point
- 'P' may be used to indicate the decimal scaling
- '0-9' indicates the number of needed digits (including overpunched).
- '#' indicates the number of optional digits.

Rounding occurs as described under Rounding in 13.6.1.1 dfdl:textNumberPattern for dfdl:textNumberRep 'standard'

13.6.2 Converting logical numbers to/from text representation

- Signed numbers with dfdl:textNumberRep 'standard' and dfdl:textStandardBase 10 are mapped using the dfdl:textNumberPattern.
- Signed numbers with dfdl:textNumberRep 'standard' and dfdl:textStandardBase not 10 are mapped to an unsigned representation. On unparsing the minimum number of characters to represent the digits is output and it is a Processing Error if the value is negative.

- Signed numbers with dfdl:textNumberRep 'zoned' are mapped using the dfdl:textNumberPattern to indicate the position of the sign and virtual decimal point. On parsing if the sign is not overpunched, that is it does not have a sign, it is treated as positive. On unparsing the sign is always overpunched.
- Unsigned numbers with dfdl:textNumberRep 'standard' and dfdl:textStandardBase 10 are mapped using the dfdl:textNumberPattern. On parsing it is a Processing Error if the data are negative.
- Unsigned numbers with dfdl:textNumberRep 'standard' and dfdl:textStandardBase not 10 are mapped to an unsigned representation. On unparsing the minimum number of characters to represent the digits is output.
- Unsigned numbers with dfdl:textNumberRep 'zoned' are mapped using the dfdl:textNumberPattern to indicate the position of the sign and virtual decimal point. On parsing it is a Processing Error if the data are negative. On unparsing the data are not overpunched with a sign.

13.7 Properties Specific to Number with Binary Representation

These properties are applicable to simple type xs:decimal and its derived types which include all the signed and unsigned integer types. These properties are not applicable to types xs:float and xs:double. See Section 13.8. Note that simple types derived from xs:decimal do not imply base-10 representations in the data stream.

<table>
<tr><th>Property Name</th><th>Description</th></tr>
<tr><td>binaryNumberRep</td><td>Enum
Valid values are 'packed', 'bcd', 'binary', 'ibm4690Packed'
Allowable values for each number type are:
<table>
<tr><th>Logical Type</th><th>Permitted Value</th></tr>
<tr><td>Decimal, Integer, NonNegativeInteger</td><td>packed, bcd, binary, ibm4690Packed</td></tr>
<tr><td>Long, Int, Short, Byte,</td><td>packed, binary, ibm4690Packed (but not bcd)</td></tr>
<tr><td>UnsignedLong, Unsignedint, UnsignedShort, UnsignedByte</td><td>packed, bcd, binary, ibm4690Packed</td></tr>
</table>

'packed' means represented as an IBM 390 packed decimal. Each byte contains two decimal digits, except for the least significant byte, which contains a sign in the least significant nibble.
'bcd' means represented as a binary coded decimal with two digits per byte.
'binary' means represented as twos complement for signed types and unsigned base-2 binary for unsigned types.

Note that the maximum allowed value for twos-complement and unsigned base-2 binary integers is implementation-dependent but MUST be at least that of a xs:long type, which is the equivalent of an 8 byte/64-bit signed integer.

'ibm4690Packed' is a variant of a packed decimal having the following characteristics:

Nibbles represent digits 0 - 9 in the usual BCD manner.
A positive value is simply indicated by digits.
A negative number is indicated by digits with the most significant nibble being xD.
If a positive or negative value packs to an odd number of nibbles, an extra xF nibble is added as the most significant nibble.

For all values, the dfdl:byteOrder property is used to determine the numeric significance of the bytes making up the representation, and the dfdl:bitOrder property is used to determine the numeric significance of the bits within a byte.
Annotation: dfdl:element, dfdl:simpleType</td></tr>
<tr><td>binaryDecimalVirtualPoint</td><td>Integer.
Used when the base simpleType is xs:decimal exactly. That is, not any of the built-in integer simple types.
An integer that represents the position of an implied decimal point within a number or specify 0.
If specified as 0 then there is no virtual decimal point
If specified as a positive integer, the position of the decimal point is moved from the least-significant side of the number toward the most-significant side of the number. For example, if 3 is specified then, the integer value 1234 represents 1.234. This is equivalent to dividing by $10^{3.}$
If specified as a negative integer, the position of the decimal point is moved from the least significant side of the number further in the less-significant direction. For example, if specified as -3, the integer value 1234 represents 1 234 000.This is equivalent to multiplying by 10^{3}.</td></tr>
</table>

	When unparsing, if the property value is not sufficient to remove the decimal point from the Infoset value, it is a Processing Error. This is true even if the resultant number can be converted into an integer (that is, all digits after the decimal point are zero) because it is an example of excess precision where no rounding is possible. Annotation: dfdl:element, dfdl:simpleType
binaryPackedSignCodes	List of Characters Used only when dfdl:binaryNumberRep or dfdl:binaryCalendarRep is 'packed' A whitespace separated string giving the hex sign nibbles to use for a positive value, a negative value, an unsigned value, and zero. Valid values for positive nibble: A, C, E, F Valid values for negative nibble: B, D Valid values for unsigned nibble: F Valid values for zero sign: A C E F 0 Example: 'C D F C' – typical S/390 usage Example: 'C D F 0' – handle special case for zero On parsing, whether to accept all valid values for a positive, negative or unsigned number, and for zero, is governed by the dfdl:binaryNumberCheckPolicy property. On unparsing, the specified values are always used. Annotation: dfdl:element, dfdl:simpleType
binaryNumberCheckPolicy	Enum Values are 'strict' and 'lax'. Indicates how lenient to be when parsing binary numbers. If 'lax' then the parser tolerates all valid alternatives where such alternatives exist. Specifically, for dfdl:binaryNumberRep 'packed' the sign nibble for positive, negative, unsigned and zero can be any of the valid respective values. On unparsing, the specified value is always used Annotation: dfdl:element, dfdl:simpleType

Table 35 Properties Specific to Number with Binary Representation

13.7.1 Converting Logical Numbers to/from Binary Representation

When unparsing a binary number (packed decimal or twos-complement) and excess precision is supplied in the Infoset no rounding occurs. It is a Processing Error.

13.7.1.1 Converting Base-2 Binary Numbers

For both parsing and unparsing, the bit string that represents the content region for a base-2 binary number is converted to/from an Infoset value by a calculation that involves the length and the dfdl:byteOrder and dfdl:bitOrder properties.

When parsing, DFDL specifies how an unsigned integer of unbounded magnitude is computed from a bit string based on its length, and the dfdl:byteOrder and dfdl:bitOrder properties. For signed types, this unbounded integer is converted into a signed value by way of the well-known twos-complement scheme, and for the xs:decimal type, the dfdl:binaryDecimalVirtualPoint property can be used to convert this integer into a decimal value with an integer and a fractional component.

A DFDL implementation can use any conversion technique consistent with this description.

13.7.1.2 Bit strings, Alignment, and dfdl:fillByte

The dfdl:alignmentUnits of 'bits', and dfdl:alignment of '1' can be used to position a bit string anywhere in the data stream without regard for any other grouping of bits into bytes.

The numeric value of the unsigned integer represented by a bit string is unaffected by alignment.

When unparsing a bit string, alignment may cause the bits within the bit string to occupy only some of the bits within a byte of the data stream. The bits of data in the alignment fill region are unspecified by the elements of

the DFDL schema, and when parsing, neither they, nor any data computed from them are put into the DFDL Infoset. During unparsing, such unspecified bits are filled in using the value of the dfdl:fillByte property. Corresponding bits from the dfdl:fillByte value are used to fill in unspecified bits of the data stream. That is, if bit K (K is 1 or greater, but less than or equal to 8) of a data stream byte is unspecified, its value is taken from bit K of the dfdl:fillByte property value.

Since the value of any bit string element is unaffected by alignment, the logical unsigned integer value for a bit-string is always computed as if the first bit were at position 1 of the bit stream. If the dfdl:length for the bit-string evaluates to M, then the bit-string conceptually occupies bits 1 to M of a data stream for purposes of computing its value.

13.7.1.3 Bits within Bit Strings of Length <= 8

Any time the length in bits, *M*, is < *8*, then when set, the bit at position *Z*, starting from the most-significant bit, (typically written on the left) supplies value *2^(M-Z)*, and the value of the bit string as an integer is the sum of these values for each of its bits.

13.7.1.4 Bits within Bit Strings of Length > 8

Call *M* the length of the bit string element in bits. In general, when *M > 8* the contribution of a bit in position *i* to the numeric value of a bit string is given by a formula specific to the dfdl:byteOrder.

For dfdl:byteOrder of 'bigEndian' the value of bit *i* is given by *2^(M - i)*, where *i = 1* is the index of the most-significant bit.

For dfdl:byteOrder of 'littleEndian' the value of bit i is given by a more complex formula. The following pseudo code computes the value of a bit in a littleEndian bit string. It is just a very big expression but is spread out over many local variables to illustrate the various sub-calculations clearly. DFDL implementations MAY use any way of converting bit strings to the corresponding integer values that is consistent with this:

In the pseudo code below:

- '%' is modular division (division where remainder is returned)
- '/' is regular division (quotient is returned)
- the expression 'a ? b : c' means 'if a is true, then the value is b, otherwise the value is c'

```
littleEndianBitValue(bitPosition, bitStringLength)
    assert bitPosition >= 1;
    assert bitStringLength >= 1;
    assert bitStringLength >= bitPosition;
    numBitsInFinalPartialByte = bitStringLength % 8;
    numBitsInWholeBytes = bitStringLength -
                          numBitsInFinalPartialByte;
    bitPosInByte = ((bitPosition - 1) % 8) + 1;
    widthOfActiveBitsInByte = (bitPosition <= numBitsInWholeBytes)
         ? 8 : numBitsInFinalPartialByte;
    placeValueExponentOfBitInByte = widthOfActiveBitsInByte -
                                    bitPosInByte;
    bitValueInByte = 2^placeValueExponentOfBitInByte;
    byteNumZeroBased = (bitPosition - 1)/8;
    scaleFactorForBytePosition = 2^(8 * byteNumZeroBased);
    bitValue = bitValueInByte * scaleFactorForBytePosition;
    return bitValue;
```

Figure 5 Little Endian bit position and value

13.7.1.4.1 Examples of Unsigned Integer Conversion

Consider the first three bytes of the data stream. Imagine their numeric values as 0x5A 0x92 0x00.

```
Positions:
00000000 01111111 11122222
12345678 90123456 78901234
Bits:
01011010 10010010 00000000
Hex values
   5   A    9   2    0   0
```

Beginning at bit position 1, (the very first bit) considering the first two bytes as a bigEndian short, the value is 0x5A92.

```
  <xs:element name="num" type="unsignedShort"
        dfdl:alignment="1"
        dfdl:alignmentUnits="bytes"
        dfdl:byteOrder="bigEndian"
        dfdl:bitOrder="mostSignificantBitFirst"
        dfdl:representation="binary"
        dfdl:binaryNumberRep="binary"/>
```

As a littleEndian short, the value is 0x925A.

```
  <xs:element name="num" type="unsignedShort"
        dfdl:alignment="1"
        dfdl:alignmentUnits="bytes"
        dfdl:byteOrder="littleEndian"
        dfdl:bitOrder="mostSignificantBitFirst"
        dfdl:representation="binary"
        dfdl:binaryNumberRep="binary"/>
```

Examining a bit string of length 13, beginning at position 2:

```
<xs:sequence>
  <xs:element name="ignored" type="unsignedByte"
        dfdl:alignment="1"
        dfdl:alignmentUnits="bits"
        dfdl:lengthUnits="bits"
        dfdl:length="1"
        dfdl:representation="binary"
        dfdl:binaryNumberRep="binary"/>
  <xs:element name="x" type="unsignedShort"
        dfdl:alignment="1"
        dfdl:alignmentUnits="bits"
        dfdl:byteOrder="bigEndian"
        dfdl:bitOrder="mostSignificantBitFirst"
        dfdl:lengthUnits="bits"
        dfdl:length="13"
        dfdl:representation="binary"
        dfdl:binaryNumberRep="binary"/>
   ...
</xs:sequence>
```

One can examine the same data stream and consider the bit positions that make up element 'x', which are the bits at positions 2 through 14 inclusive.

```
Positions:
00000000 01111111 11122222
12345678 90123456 78901234
Bits:
 1011010 100100
```

Since alignment does not affect logical value, one obtains the same logical value as if the bits were realigned. That is, the value is the same as if the bits of the element's representation began with bit position 1.

```
Realigned Positions:
00000000 01111111 11122222
12345678 90123456 78901234
Bits:
10110101 00100
```

The DFDL schema fragment above gives element 'x' the dfdl:byteOrder 'bigEndian' property and the dfdl:bitOrder 'mostSignificantBitFirst' property. In this case the place value of each position is given by $2^{(M-i)}$. Below the bit values are lined up underneath their place-values.

```
Place value of bits
...11110 00000000
...21098 76543210
Bit values
...10110 10100100
Hex values
   1   6    A   4
```

The value of element 'x' is 0x16A4. Notice how it is the most-significant byte -- which is the first byte when big endian -- that becomes the partial byte (having fewer than 8 bits) in the case where the length of the bit string is not a multiple of 8 bits.

For dfdl:byteOrder of 'littleEndian'. The place values of the individual bits are not as easily visualized. However there is still a basic formula (given in the pseudo code in Section 13.7.1.4 Bits within Bit Strings of Length > 8) and value.

Looking again at our realigned positions:

```
Realigned Positions:
00000000 01111111 11122222
12345678 90123456 78901234
Bits:
10110101 00100
```

The place values of each of these bits, for little endian byte order can be seen to be:

```
PlaceValue positions
00000000 ...11100
76543210 ...21098
Bit values
10110101 ...00100
Hex values
   B    5    0    4
```

One must reorder the bytes for little endian byte order. The value of element 'x' is 0x04B5. In little endian form, the first 8 bits make up the first byte, and that contains the least-significant byte of the logical numeric unsignedShort value. The additional bits of the partial byte are once again the most significant byte; however, for little endian form, this is the second byte. The second byte contains only 5 bits, and they are the most significant bits within that byte, but they are treated as if shifted to become the least significant 5 bits of a logical byte that contributes to the integer value. This logical byte makes up the most-significant byte of the unsignedShort integer.

Now examine the 13 bits beginning at position 2, in the context where dfdl:byteOrder is 'littleEndian' and dfdl:bitOrder is 'leastSignificantBitFirst' and dfdl:binaryNumberRep is 'binary'.

In this case, the bit positions are assigned differently. Below the bytes are shown left-to-right:

```
Positions:
00000000 11111110 22222111
87654321 65432109 43210987
Bits:
0101101 0 10010010 00000000
Hex values
   5   A    9   2    0   0
```

The bits of interest are highlighted above. Redisplaying this same data, but reversing the order of the bytes to right-to-left, then one gets:

```
Positions:
22222111 11111110 00000000
43210987 65432109 87654321
Bits:
00000000 10010010 0101101 0
Hex values
   0   0    9   2    5   A
```

The above shows more clearly the contiguous region of bits containing:

```
0 1001 0010 1101
```

or the value 0x092D.

13.7.1.5 **Converting Packed Decimal Numbers**

Signed numbers with dfdl:binaryNumberRep 'packed' are parsed using a nibble to indicate the sign. The unsigned nibble is treated as positive. On unparsing the sign nibble is written according to dfdl:binaryPackedSignCodes. The unsigned nibble is never written.

Signed numbers with dfdl:binaryNumberRep 'bcd' are always positive. On unparsing it is a Processing Error if the Infoset data is negative.

Signed numbers with dfdl:binaryNumberRep 'ibm4690Packed' are parsed using the sign nibble to identify negative values. There is no sign nibble for positive values. On unparsing the nibble 0xD is written for negative values.

Unsigned numbers with dfdl:binaryNumberRep 'packed' are parsed if the nibble is positive or unsigned. It is a Processing Error if the data is negative. On unparsing the unsigned nibble is used.

Unsigned numbers with dfdl:binaryNumberRep 'bcd' are readily parsed as BCD data is always positive.

Unsigned numbers with dfdl:binaryNumberRep 'ibm4690Packed' are parsed if there is no sign nibble of 0xD to identify a negative value. It is a Processing Error if the data is negative. On unparsing no sign nibble is written.

13.8 Properties Specific to Float/Double with Binary Representation

Property Name	Description
binaryFloatRep	Enum or DFDL Expression This specifies the encoding method for the float and double. Valid values are 'ieee', 'ibm390Hex',This property can be computed by way of an expression which returns a string of 'ieee' or ' ibm390Hex' . The expression must not contain forward references to elements which have not yet been processed. The enumeration value 'ieee' refers to the IEEE 754-1985 specification. For both 'ieee' and 'ibm390hex', an xs:float must have a physical length of 4 bytes. It is a Schema Definition Error if there is a specified length not equivalent to 4 bytes. Similarly, for both 'ieee' and 'ibm390hex', an xs:double must have a physical length of 8 bytes. It is a Schema Definition Error if there is a specified length not equivalent to 8 bytes. The dfdl:byteOrder property is used to construct a value from the bytes in the binary representation. Note: The DFDL Infoset float and double data types match the precision of the IEEE specification. There may be precision/rounding issues when converting IBM float/double to/from the DFDL Infoset float/double types. Half-precision IEEE and quad-precision IEEE/IBM are not supported.[43] Annotation: dfdl:element, dfdl:simpleType

Table 36 Properties Specific to Float/Double with Binary Representation

13.9 Properties Specific to Boolean with Text Representation

Property Name	Description
textBooleanTrueRep	List of DFDL String Literals or DFDL Expression A whitespace separated list of representations to be used for 'true'. These are compared after trimming when parsing, and before padding when unparsing. If dfdl:lengthKind is 'explicit' or 'implicit' and either dfdl:textPadKind or dfdl:textTrimKind is 'none' then both dfdl:textBooleanTrueRep and dfdl:textBooleanFalseRep must have the same length else it is a Schema Definition Error. This property can be computed by way of an expression which returns a string of whitespace separated list of values. The expression must not contain forward references to elements which have not yet been processed. On unparsing the first value is used If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. *Text Boolean Character Restrictions:* The string literal is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is not allowed. • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are not allowed It is a Schema Definition Error if the string literal is the empty string or contains any of the disallowed constructs. Annotation: dfdl:element, dfdl:simpleType
textBooleanFalseRep	List of DFDL String Literals or DFDL Expression

[43] Note that XSD 1.1 moved to IEEE 754-2008 only because of new decimal support, and not for enhanced float support. That's why in XSD 1.1 there are still just the xs:float and xs:double built-in types. Any future support for half-precision and quad-precision in XSD would very likely be implemented by adding new built-in types that derive from xs:anySimpleType. It is likely therefore that future DFDL support for half-precision and quad-precision will build on XSD.

	A whitespace separated list of representations to be used for 'false' These are compared after trimming when parsing, and before padding when unparsing. If dfdl:lengthKind is 'explicit' or 'implicit' and either dfdl:textPadKind or dfdl:textTrimKind is 'none' then both dfdl:textBooleanTrueRep and dfdl:textBooleanFalseRep must have the same length else it is a Schema Definition Error. This property can be computed by way of an expression which returns a string of whitespace separated list of values. The expression must not contain forward references to elements which have not yet been processed. On unparsing the first value is used If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. The string literal value is restricted in the same way as described in "Text Boolean Character Restrictions" in the description of the dfdl:textBooleanTrueRep property. Annotation: dfdl:element, dfdl:simpleType
textBooleanJustification	Enum Valid values 'left', 'right', 'center' Controls how the data is padded or trimmed on parsing and unparsing. Behavior as for dfdl:textStringJustification. Annotation: dfdl:element, dfdl:simpleType
textBooleanPadCharacter	DFDL String Literal The value that is used when padding or trimming boolean elements. The value can be a single character or a single byte. If a character, then it can be specified using a literal character or using DFDL entities. If a byte, then it must be specified using a single byte value entity. If a pad character is specified when lengthUnits is 'bytes' then the pad character must be a single-byte character. If a pad byte is specified when lengthUnits is 'characters' then • the dfdl:encoding must be a fixed-width encoding • padding and trimming must be applied using a sequence of N pad bytes, where N is the width of a character in the fixed-width encoding. The string literal value is restricted in the same way as described in "Pad Character Restrictions" in the description of the dfdl:textStringPadCharacter property. Annotation: dfdl:element, dfdl:simpleType

Table 37 Properties Specific to Boolean with Text Representation

13.10 Properties Specific to Boolean with Binary Representation

Property Name	Description
binaryBooleanTrueRep	Non-negative Integer This value, treated as a binary xs:unsignedInt (See Section 13.7.1 Converting Logical Numbers to/from Binary Representation), gives the representation to be used for 'true' If this property value is the empty string, when parsing it means dfdl:binaryBooleanTrueRep is any value other than dfdl:binaryBooleanFalseRep; when unparsing, the one's complement of the dfdl:binaryBooleanFalseRep is used. The length of the data value of the element must be between 1 bit and 32 bits (4 bytes) as described in Section 12.3.7.2. It is a Schema Definition Error if the value (when provided) of dfdl:binaryBooleanTrueRep cannot fit as an unsigned binary integer in the specified length.

	Annotation: dfdl:element, dfdl:simpleType
binaryBooleanFalseRep	Non-negative Integer This value, treated as a binary xs:unsignedInt (See Section 13.7.1 Converting Logical Numbers to/from Binary Representation), gives the representation to be used for 'false' The length of the data value of the element must be between 1 bit and 32 bits (4 bytes) as described in Section 12.3.7.2. It is a Schema Definition Error if the value of dfdl:binaryBooleanFalseRep cannot fit as an unsigned binary integer in the specified length. Annotation: dfdl:element, dfdl:simpleType

Table 38 Properties Specific to Boolean with Binary Representation

13.11 Properties Specific to Calendar with Text or Binary Representation

The properties describe how a calendar (that is, date/time data) is to be interpreted including an unparsing pattern property plus properties that qualify the pattern.

These properties can be used when a calendar has dfdl:representation 'text' or dfdl:representation 'binary' and a packed decimal representation.

Property Name	Description
calendarPattern	String Defines the ICU pattern that describes the format of the calendar. The pattern defines where the year, month, day, hour, minute, second, fractional second and time zone components appear. See calendarPattern property section below. When the dfdl:representation is 'binary' and the representation is a packed decimal then the pattern can contain only characters and symbols that always result in the presentation of digits. Annotation: dfdl:element, dfdl:simpleType
calendarPatternKind	Enum Valid values 'explicit', 'implicit' 'explicit' means the pattern is given by dfdl:calendarPattern, 'implicit' means the pattern is derived from the XML schema date/time type. **Logical Type** / **Default Pattern** xs:date / yyyy-MM-dd xs:dateTime / yyyy-MM-dd'T'HH:mm:ss xs:time / HH:mm:ssZ Annotation: dfdl:element, dfdl:simpleType
calendarCheckPolicy	Enum Valid values are 'strict', 'lax' Indicates how lenient to be when parsing against the pattern. See Section 13.11.2 The dfdl:calendarCheckPolicy Property below for details of the specific behaviors for 'strict' and 'lax'. Annotation: dfdl:element, dfdl:simpleType
calendarTimeZone	String This property provides the time zone that is assumed if no time zone explicitly occurs in the data. Valid values specify a UTC time zone offset by matching the regular expression: `(UTC)([+\-]([01]\d|\d)((([:][0-5]\d){1,2})?))?`

	In addition, empty string can be specified to indicate "no time zone" which simply leaves the time zone unknown/unspecified. Data which does not specify a time zone does not obtain a time zone from this property and so simply lacks time zone information. The IANA time zone format (also known as the Olson time zone format) may also be used. (e.g., America/New_York)) See [IANATimeZone]. Note that this property is used when parsing only. Annotation: dfdl:element, dfdl:simpleType
calendarObserveDST	Enum Valid values are 'yes', 'no' Whether the time zone given in dfdl:calendarTimeZone observes daylight savings time. Ignored if dfdl:calendarTimeZone is specified in UTC format, or if dfdl:calendarTimeZone is empty string. That is, this property is used only if the dfdl:calendarTimeZone is in IANA (also known as Olson) format [IANATimeZone]. This property applies to parsing only. Annotation: dfdl:element, dfdl:simpleType
calendarFirstDayOfWeek	Enum Valid values 'Monday' … 'Sunday' The day of the week upon which a new week is considered to start. Annotation: dfdl:element, dfdl:simpleType
calendarDaysInFirstWeek	Non-negative Integer Valid values 1 to 7 Specify the number of days of the new year that must fall within the first week. The start of a year usually falls in the middle of a week. If the number of days in that week is less than the value specified here, the week is considered to be the last week of the previous year; hence week 1 starts some days into the new year. Otherwise it is considered to be the first week of the new year; hence week 1 starts some days before the new year. Annotation: dfdl:element, dfdl:simpleType
calendarCenturyStart	Non-negative Integer Valid values 0 to 99. This property determines on parsing how two-digit years are interpreted. It specifies the two digits that start a 100-year window that contains the current year. For example, if 89 is specified, and the current year is 2006, all two-digit dates are interpreted as being in the range 1989 to 2088. A two-digit year less than 89 is interpreted as 20nn and a two-digit year more than or equal to 89 is treated as 19nn. Annotation: dfdl:element, dfdl:simpleType
calendarLanguage	String or DFDL Expression The language that is used when the pattern produces a presentation in text such as for names of the months, and names of days of the week. The value must match the regular expression: `([A-Za-z]{1,8}([\-_][A-Za-z0-9]{1,8})*)` It is a Schema Definition Error otherwise. The expression must not contain forward references to elements which have not yet been processed. All DFDL Implementations MUST support dfdl:calendarLanguage value "en".

	DFDL implementations MAY support additional values, however, the value of the dfdl:calendarLanguage property is always interpreted as a Unicode Language Identifier as defined by [LDML], and [CLDR]. Annotation: dfdl:element, dfdl:simpleType

Table 39 Properties specific to Calendar with Text or Binary Representation

13.11.1 The dfdl:calendarPattern property

The dfdl:calendarPattern describes how to parse and unparse text and binary representations of dateTime, date and time logical types. The pattern is primarily used on unparsing to define the format but is also used to aid parsing.

The pattern is derived from the ICU SimpleDatetimeFormat class described here: [ICUDateTime], which uses symbols defined by [LDML].

An extension is the formatting symbol I which means accept a subset of ISO 8601 [ISO8601] compliant calendars

Symbol	Presentation	Meaning	Example	
G	Text	era designator	G	AD
y	Number	year	y	1996
			yyyy	1996
			yy	96
u	Number	year(allows negative years)	u	1900, 0, -500
Y	Number	year (of the week of year)	Y	1997
M	Text & Number	month in year	M	9, 12
			MM	09, 12
			MMM	Sep
			MMMM	September
			MMMMM	S
d	Number	day in month	d	2
			dd	02
h	Number	hour in am/pm (1~12)	h	7
			hh	07
H	Number	hour in day (0~23)	H	0
			HH	00
m	Number	minute in hour	m	4
			mm	04
s	Number	second in minute	s	5
			ss	05
S	Number	fractional second	S	2
			SS	23
			SSS	235
E	Text	day of week	E	Tue
			EE	Tue
			EEE	Tue

			EEEE	Tuesday
			EEEEE	T
			EEEEEE	Tu
e	Text & Number	day of week (local)	e	2
			ee	02
			eee	Tue
			eeee	Tuesday
			eeeee	T
			eeeeee	Tu
D	Number	day in year	D	3, 24, 189
			DD	03, 24, 189
			DDD	003, 024, 189
F	Number	day of week in month	F	2 (2nd Wed in July)
w	Number	week in year	w, ww	27
W	Number	week in month	W	2
a	Text	am/pm marker	A	pm
k	Number	hour in day (0~24)	k	2, 24
			kk	02, 24
K	Number	hour in am/pm (0~11)	K	0
			KK	00
z	Text	time zone: specific non-location	z, zz, zzz	PDT
			zzzz	Pacific Daylight Time
Z	Text	time zone: ISO8601 basic format	Z, ZZ, ZZZ	-0800, +0000
		time zone: long localized GMT	ZZZZ	GMT-08:00, GMT+00:00
O	Text	time zone: localized GMT	O	GMT-8
			OOOO	GMT-08:00
v	Text	time zone: generic non-location	v	PT
			vvvv	Pacific Time
V	Text	time zone: short time zone ID	V	uslax
		time zone: long time zone ID	VV	America/Los_Angeles
		time zone: exemplar city	VVV	Los Angeles
		time zone: generic location.	VVVV	Los Angeles Time
x	Text	time zone: ISO8601 basic or extended format	x	-08, +0530, +0000
			xx	-0800, +0000
			xxx	-08:00, +00:00
X	Text		X	-08, +0530, Z

		Time Zone: ISO8601 basic or extended format .The UTC indicator "Z" is used when local time offset is 0.	XX	-0800, Z
			XXX	-08:00, Z
l	Text	ISO8601 date/time	l	2006-10-07T12:06:56.568+01:00
'	Delimiter	escape for text	'	'Date='
''	Literal	single quote	''	'o''clock'

Table 40 Symbols in the dfdl:calendarPattern Property

The count of pattern letters determines the format as indicated in the table.

When numeric fields abut one another directly, with no intervening delimiter characters, they constitute a run of abutting numeric fields. Such runs are parsed specially as described at [ICUDateTime].

The maximum number of "S" symbols that may appear in the pattern is implementation-defined but MUST be at least three. The stored accuracy for fractional seconds is also implementation-defined but MUST be at least millisecond accuracy. When the number of "S" symbols in a pattern exceeds the supported accuracy, excess fractional seconds are truncated from the right (not rounded) when parsing, and zeros are added to the right when unparsing. For example, a DFDL processor allows up to six "S" symbols and has millisecond accuracy; for pattern "ss.SSSSSS", data "12.345678" would be parsed into Infoset xs:time "00:00:12:345", which would be unparsed into data "12.345000".

Unlike other fields, fractional seconds, “S”, are padded on the right with zero.

It is a Processing Error if seconds appear in that part of the SimpleContent region that represents a time zone.

The count of pattern letters determines the format as indicated in the table.

If dfdl:representation is text, any characters in the pattern that are not in the ranges of ['a'..'z'] and ['A'..'Z'] are treated as quoted text. For instance, characters like ':', '.', ' ', '#' and '@' appear in the formatted output even if they are not embraced within single quotes. The single quote is used to 'escape' letters. Two single quotes in a row, whether inside or outside a quoted sequence, represent a 'real' single quote.

If dfdl:representation is binary, then the pattern can contain only characters and symbols that always result in the presentation of digits.

The symbols 'z', 'zz', and 'zzz' have identical meaning, as do 'Z', 'ZZ', and 'ZZZ'.

The 'l' symbol must not be used with any other symbol except for 'escape for text'. It represents calendar formats that match those defined in the restricted profile of the ISO 8601 standard proposed by the W3C at http://www.w3.org/TR/NOTE-datetime. The formats are referred to as 'granularities'.

- xs:dateTime. When parsing, the data must match one of the granularities. When unparsing, the fullest granularity is used.
- xs:date. When parsing, the data must match one of the date-only granularities. When unparsing, the fullest date-only granularity is used.
- xs:time. When parsing, the data must match only the time components of one of the granularities that contains time components. When unparsing, the time components of the fullest granularity are used. The literal 'T' character is not expected in the data when parsing and is not output when unparsing.
- The number of fractional second digits supported is the same as for the “S” fractional seconds specifier described above.
- The omission of time zone from the input data when the type is xs:dateTime or xs:time is not a Processing Error. If that occurs then the time zone is obtained from the calendarTimeZone property.
- When unparsing and the time zone is UTC, the time zone is output as ‘+00:00’.

When parsing, for any pattern that omits components the values for the omitted components are supplied from the Unix epoch 1970-01-01T00:00:00.000.[44]

[44] Note that DFDL does not support an isolated month, day, or year that is not part of a greater date type, as it does not support the XSD simple types xs:gMonth, xs:gDay, and xs:gYear.

When unparsing, and the pattern contains a formatting symbol that requires a component of the date/time and the Infoset value does not contain that component, it is a Processing Error.

When parsing a calendar element with a packed decimal representation then the nibbles from the data are converted to text digits without any trimming of leading or trailing zeros, and the result is then matched against the pattern according to the usual rules.

When unparsing, if a time zone symbol is not available for a particular time zone, a fallback may be used as defined in [ICUDateTime].

13.11.2 **The dfdl:calendarCheckPolicy Property**

The differences in behavior between 'strict' and 'lax' for this property can be subtle. Both are quite lenient in enforcement of many variations in format, with the 'lax' value adding additional tolerance of more format variations to those already allowed by the 'strict' value.

1. Lenient parsing behaviour when in 'strict' policy:
    a. Case insensitive matching for text fields
    b. MMM, MMMM, and MMMMM all accept either short or long form of Month
    c. E, EE, EEE, EEEE, EEEEE , and EEEEEE all accept either abbreviated, full, narrow and short forms of Day of Week
    d. Accepts truncated leftmost numeric field (e.g., pattern "HHmmss" allows "123456" (12:34:56) and "23456" (2:34:56) but not "3456")
2. Additional lenient parsing behaviour when in 'lax' policy is implementation-defined, but typically includes:
    a. Values outside valid ranges are normalized (e.g., "March 32 1996" is treated as "April 1 1996")
    b. Ignoring a trailing dot after a non-numeric field
    c. Leading and trailing whitespace in the data but not in the pattern is accepted
    d. Whitespace in the pattern can be missing in the data
    e. Partial matching on literal strings. E.g., data "20130621d" allowed for pattern "yyyyMMdd'date'"

13.12 Properties Specific to Calendar with Text Representation

Property Name	Description
textCalendarJustification	Enum Valid values 'left', 'right', 'center' Controls how the data is padded or trimmed on parsing and unparsing. Behavior as for dfdl:textStringJustification. Annotation: dfdl:element, dfdl:simpleType
textCalendarPadCharacter	DFDL String Literal The value that is used when padding or trimming calendar elements. The value can be a single character or a single byte. If a character, then it can be specified using a literal character or using DFDL entities. If a byte, then it must be specified using a single byte value entity If a pad character is specified when dfdl:lengthUnits is 'bytes' then the pad character must be a single-byte character. If a pad byte is specified when dfdl:lengthUnits is 'characters' then • the encoding must be a fixed-width encoding • padding and trimming must be applied using a sequence of N pad bytes, where N is the width of a character in the fixed-width encoding. The string literal value is restricted in the same way as described in "Pad Character Restrictions" in the description of the dfdl:textStringPadCharacter property. Annotation: dfdl:element, dfdl:simpleType

Table 41 Properties Specific to Calendar with Text Representation

13.13 Properties Specific to Calendar with Binary Representation

Property Name	Description
binaryCalendarRep	Enum Valid values are 'packed', 'bcd', 'ibm4690Packed', 'binarySeconds', 'binaryMilliseconds' For all values, the dfdl:byteOrder property is used to determine the numeric significance of the bytes making up the representation. • 'packed' means represented as an IBM 390 packed decimal. Each byte contains two decimal digits, except for the rightmost byte, which contains a sign to the right of a decimal digit. The digits are interpreted according to the dfdl:calendarPattern property. Property dfdl:binaryPackedSignCodes is applicable. • 'bcd' means represented as a binary coded decimal with two digits per byte. The digits are interpreted according to the dfdl:calendarPattern property • 'ibm4690Packed' means represented as a variant of packed format as described in property dfdl:binaryNumberRep. The digits are interpreted according to the dfdl:calendarPattern property. For all packed decimals, property dfdl:binaryNumberCheckPolicy is applicable. For all these packed decimals, dfdl:calendarPattern can contain only characters and symbols that always result in the presentation of digits. It is a Schema Definition Error otherwise. This implies that property dfdl:calendarPatternKind must be 'explicit' because the default patterns for 'implicit' contain non-numeric characters. It is a Schema Definition Error otherwise. See Section 13.7 Properties Specific to Number with Binary Representation. Note also that a virtual decimal point for the boundary between seconds and fractional seconds is implied from the pattern at the boundary of 's' and 'S', i.e., where the substring 'sS' appears in the pattern. • 'binarySeconds' means represented as binary xs:int, that is, as a 4 byte signed integer that is the number of seconds from the epoch (positive or negative). It is a Schema Definition Error if there is a specified length not equivalent to 4 bytes. • 'binaryMilliseconds' means represented as binary xs:long, that is, as an 8 byte signed integer that is the number of milliseconds from the epoch (positive or negative). It is a Schema Definition Error if there is a specified length not equivalent to 8 bytes. Values binarySeconds and binaryMilliseconds may only be used when the type is xs:dateTime. (It is a Schema Definition Error otherwise.) Annotation: dfdl:element, dfdl:simpleType
binaryCalendarEpoch	DateTime Used when dfdl:binaryCalendarRep is 'binarySeconds' or 'binaryMilliseconds' The epoch from which to calculate dates and times. If the time zone is omitted from the value, then UTC is used. Annotation: dfdl:element, dfdl:simpleType

Table 42 Properties Specific to Calendar with Binary Representation

Examples of packed decimal format calendars for December 14, 1923 and dfdl:calendarPattern of 'MMddyy' would be:

- packed: (hexadecimal) `01 21 42 3C`
- bcd: (hexadecimal) `12 14 23`
- ibm4690Packed: (hexadecimal) `12 14 23`

The 'C' nibble at the end of the 'packed' representation is a sign nibble, and the leading 0 nibble is just to align to a byte boundary..

13.14 Properties Specific to Opaque Types (xs:hexBinary)

There are no properties specific to opaque types

13.15 Nil Value Processing

Sometimes it is desirable to represent an unused element, place-holder for unknown information, or inapplicable information *explicitly* with an element, rather than by the lack of an element.

For example, it may be desirable to represent a sparsely populated array of data using a distinguished nil element to fill the locations where data is absent, thereby preserving the position for the elements that are present.

As another example, it may be desirable to represent an unused simple element by a value which is not conformant to the logical type of the element.

Such cases can be represented using the DFDL nil mechanism which is based on the XML Schema nil mechanism. DFDL provides what are commonly called "in-band" nil values by way of dfdl:nilKind 'logicalValue', and also provides for two kinds of literal indicators of nil through dfdl:nilKind 'literalValue' and dfdl:nilKind 'literalCharacter'. Nil processing is used when the XSD 'nillable' property of an element is true.

DFDL allows elements of complex type to be nillable. However, to avoid the concept of a complex element having a value, which does not exist in DFDL, the only permissible nil value is the empty string, represented by the DFDL %ES; entity and using dfdl:nilKind 'literalValue'.

On parsing, an element occurrence is nil if the element has XSD nillable 'true' and the data is a nil representation as defined in Section 9.2.1. Specifically:

1. When dfdl:nilKind is 'literalValue', the ***NilLiteralValue*** region of the data stream matches any of the dfdl:nilValue values.
2. When dfdl:nilKind is 'literalCharacter', all characters in the ***NilLiteralCharacters*** region of the data stream match the dfdl:nilValue character.
3. When dfdl:nilKind is 'logicalValue', the data contains a normal representation, and the ***NilLogicalValue*** region of the data stream, converted to the element's logical type, matches any of the dfdl:nilValue values.

For dfdl:nilKind 'literalValue' or 'literalCharacter':

- Determination of whether the data is a nil representation for a literal nil happens first before any consideration of whether the representation is the empty, normal, or absent representations.
- Property dfdl:nilValueDelimiterPolicy controls whether matching one of the nil values also involves matching the initiator or terminator specified by the element. This gives control over whether a nil indicator may or may not also require the delimiters that a normal data element requires.

On unparsing, an element is nil if XSD nillable is 'true' AND the element information item in the augmented Infoset has the **[nilled]** member as true, in which case what is output to the data stream is one of the following:

1. When dfdl:nilKind is 'logicalValue' then the first value of dfdl:nilValue converted to the physical representation is output as the ***NilLogicalValue*** region.
2. When dfdl:nilKind is 'literalValue' then the first value of dfdl:nilValue is output as the ***NilLiteralValue*** region.
3. When dfdl:nilKind is 'literalCharacter' then the character from dfdl:nilValue, repeated to the needed length, is output as the ***NilLiteralCharacters*** region.

For dfdl:nilKind 'literalValue' or 'literalCharacter' then dfdl:nilValueDelimiterPolicy determines whether any initiator or terminator also appear surrounding the literal nil in the output data.

13.16 Properties for Nillable Elements

These properties are used when the XSD 'nillable' property of an element is 'true', and they control when and how the representation data are interpreted as having the logical meaning 'nil'.

Property Name	Description
nilKind	Enum Valid values 'literalValue', 'logicalValue', 'literalCharacter'. Used when XSD nillable is 'true'.

	Specifies how dfdl:nilValue is interpreted to represent the nil value in the data stream. If 'literalCharacter' then dfdl:nilValue specifies a single character or a single byte that, when repeated to the length of the element, is the nil value. 'literalCharacter' may only be specified for fixed-length elements, otherwise it is a Schema Definition Error.. If 'literalValue' then dfdl:nilValue specifies a list of DFDL literal strings that are the possible representations for nil. If 'logicalValue' then dfdl:nilValue specifies a list of logical values that are the possible logical values for nil. Complex elements can be nillable, but dfdl:nilKind can only be 'literalValue' and dfdl:nilValue must be "%ES;". It is a Schema Definition Error otherwise. Annotation: dfdl:element
nilValue	List of DFDL String Literals, List of Logical Values, DFDL String Literal Specifies the text strings that are the possible literal or logical nil values of the element. If dfdl:nilKind is 'literalValue' then dfdl:nilValue specifies a whitespace separated list of DFDL literal strings that are the possible representations for nil. On parsing the element value is nil if the trimmed data matches one of the string literals in the list. On unparsing if the element value is nil the first string literal in the list is output. If dfdl:nilKind is 'logicalValue' then dfdl:nilValue specifies a whitespace separated list of logical values that are the possible logical values for nil. On parsing the element value is nil if the data, converted to its logical type, matches any of the logical values in the list. On unparsing if the element value is nil, the first value from the list is converted to its physical representation and output. If dfdl:nilKind is 'literalCharacter' then dfdl:nilValue specifies a single character or byte that, when repeated to the length of the element, is the nil representation. If a character, then it can be specified using a literal character or using DFDL entities. If a character is specified when dfdl:lengthUnits is 'bytes' then the dfdl:nilValue must be a single-byte character. To specify a byte, it must be specified using a single "%#r;" entity. If a byte is specified when dfdl:lengthUnits is 'characters' then the dfdl:encoding must be a fixed-width encoding. On parsing, the element value is nil if all characters in the untrimmed data content match the dfdl:nilValue character . On unparsing, if the element value is nil the dfdl:nilValue character is output to the needed length. There are restrictions on the string literal syntax of dfdl:nilValue. When dfdl:nilKind is literalValue and text representation: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is allowed • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are allowed. When dfdl:nilKind is literal value and binary representation: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is allowed • DFDL Character class ES is allowed. • Other DFDL Character classes NL, WSP, WSP+, and WSP*, are not allowed. When dfdl:nilKind is literalCharacter and text representation: • DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is allowed. • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are not allowed. When dfdl:nilKind is literalCharacter and binary representation:

	• DFDL character entities are allowed • The DFDL byte value entity (%#rXX;) is allowed • DFDL Character classes NL, WSP, WSP+, WSP*, and ES are not allowed. dfdl:nilValue is sensitive to dfdl:ignoreCase when dfdl:nilKind is 'literalValue' or 'logicalValue', but not when dfdl:nilKind is 'literalCharacter' Complex elements can be nillable, but dfdl:nilKind can only be 'literalValue' and dfdl:nilValue must be "%ES;". It is a Schema Definition Error otherwise. Annotation: dfdl:element
nilValueDelimiterPolicy	Enum Valid values are 'none', 'initiator', 'terminator' or 'both'. Indicates that when the value nil is represented, an initiator (if one is defined), a terminator (if one is defined), both an initiator and a terminator (if defined) or neither must be present. This property enables distinguishing the nil representation from the representation of a value or an empty representation based on presence or absence of the initiator and terminator. Ignored if both dfdl:initiator and dfdl:terminator are "" (empty string). Ignored if dfdl:nilKind is set to 'logicalValue' In this case the DFDL processor treats a nil representation like any other representation of the element in that it expects delimiters when parsing, outputs them when unparsing. 'initiator' indicates that, on parsing, the dfdl:initiator followed by a dfdl:nilValue indicates that a nil representation is present. It also indicates that on unparsing when the logical value is nil that the dfdl:initiator is output followed by the first dfdl:nilValue. 'terminator' indicates that, on parsing, a dfdl:nilValue followed by the dfdl:terminator indicates that a nil representation is present. It also indicates that on unparsing when the logical value is nil the first dfdl:nilValue followed by the dfdl:terminator is output. 'both' indicates that, on parsing, both the dfdl:initiator and dfdl:terminator must be present with a dfdl:nilValue to indicate that a nil representation is present. On unparsing the dfdl:initiator followed by the first dfdl:nilValue, followed by the dfdl:terminator is output. 'none' indicates that a dfdl:nilValue without any dfdl:initiator or dfdl:terminator indicates that a nil representation is present. On unparsing the first dfdl:nilValue is output without any dfdl:initiator or dfdl:terminator. The value of dfdl:nilValueDelimiterPolicy MUST only be checked if there is a dfdl:initiator or dfdl:terminator in scope. If so, and dfdl:nilValueDelimiterPolicy is not set, it is a Schema Definition Error. If dfdl:initiator is not "" and dfdl:terminator is "" and dfdl:nilValueDelimiterPolicy is 'terminator' it is a Schema Definition Error. If dfdl:terminator is not "" and dfdl:initiator is "" and dfdl:nilValueDelimiterPolicy is 'initiator' it is a Schema Definition Error. It is not a Schema Definition Error if dfdl:nilValueDelimiterPolicy is 'both' and one or both of dfdl:initiator and dfdl:terminator is "". This is to accommodate the common use of setting 'both' as a schema-wide setting. It is a Schema Definition Error if dfdl:nilValueDelimiterPolicy is set to 'none' or 'terminator' when the parent xs:sequence has dfdl:initiatedContent 'yes'. Annotation: dfdl:element
useNilForDefault	Enum Valid values are 'yes', 'no' When the conditions for applying a simple element default are satisfied, this property controls whether to set the Infoset item **[nilled]** boolean member, or to use the XSD default or fixed properties to obtain a data value.

	This property has precedence over the XSD default and XSD fixed properties. It is only used, and must be defined, if the XSD nillable property is 'true'. Defaulting occurs as described in Section 9.4 Element Defaults with nil as the default value. The dfdl:nilValue property must specify at least one nil value otherwise it is a Schema Definition Error. The dfdl:nilKind property may be any of its values. Annotation: dfdl:element (simpleType)

Table 43 Properties for Nillable Elements

The DFDL element defaults processing uses XSD default, XSD fixed or dfdl:useNilForDefault to provide a default value. See Section 9.4 Element Defaults for a full description.

14 Sequence Groups

The following properties are specific to sequences.

Property Name	Description
sequenceKind	Enum Valid values are 'ordered', 'unordered' When 'ordered', this property means that the contained items of the sequence are expected in the same order that they appear in the schema, which is called schema-definition-order. When 'unordered', this property means that the items of the sequence are expected in any order. Repeating occurrences of the same element do not need to be contiguous. The children of an unordered sequence must be xs:element otherwise it is a Schema Definition Error. Annotation: dfdl:sequence, dfdl:group (sequence)
initiatedContent	Enum Valid values are 'yes', 'no' When 'yes' indicates that all the children of the sequence are initiated. It is a Schema Definition Error if any children have their dfdl:initiator property set to the empty string. If the child is optional then it is known to exist when its initiator has been found. Any subsequent error parsing the child does not cause the parser to backtrack to try other alternatives. When 'no', the children of the sequence may have their dfdl:initiator property set to the empty string. Annotation: dfdl:sequence, dfdl:choice, dfdl:group

Table 44 Properties for Sequence Groups

A sequence can have a dfdl:initiator and/or a dfdl:terminator as described earlier.

14.1 Empty Sequences

A sequence having no children is syntactically legal in DFDL. In the data stream, such a sequence can have non-zero length ***LeftFraming*** and ***RightFraming*** regions, but the SequenceContent region in between must be empty. It is a Processing Error if the SequenceContent region of an empty sequence has non-zero length when parsing.

XML schema does not define an empty sequence that is the content model of a complex type definition as effective content so any DFDL annotations on such a construct would be ignored. It is a Schema Definition Error if the empty sequence is the content model of a complex type, or if a complex type has nothing in its content model at all.

A hidden group reference is indicated in DFDL using an empty sequence such as

```
<xs:sequence dfdl:hiddenGroupRef="QName"/>
```

To XML Schema this is an empty sequence group; hence it is a Schema Definition Error if this appears as the model group of a complex type. Otherwise this is not considered an empty sequence, but a group reference.

14.2 Sequence Groups with Separators

Additional properties apply to sequence groups that use text delimiters to separate one occurrence of a member of the group from the next. Such a delimiter is called a separator. DFDL provides several properties that control the parsing and writing of separators, and satisfy the requirement to model sequences where:

1. A separator has alternative potential representations in the data.
2. A separator is placed before, after, or between occurrences in the data.
3. Separators are used to indicate the position of occurrences in the data

These requirements are addressed by the properties dfdl:separator, dfdl:separatorPosition and dfdl:separatorSuppressionPolicy, as described below.

These properties combine to define the syntax for a sequence group with dfdl:sequenceKind 'ordered'. Not all combinations of the properties give rise to consistent syntax, so some combinations are disallowed and give rise to a Schema Definition Error.

In some sequences, the presence of separators alone is enough to establish occurrences within the sequence. Such a sequence is called a *positional* sequence.

Positional sequence - Each occurrence in the sequence can be identified by its position in the data. Typically, the components of such a sequence do not have an initiator. In some such sequences, the separators for optional zero-length occurrences may or must be omitted when at the end of the group. In DFDL, a sequence is considered positional if it contains only required elements and/or optional and array elements that have dfdl:occursCountKind 'implicit', 'fixed' or 'expression', and it has dfdl:separatorSuppressionPolicy 'never', 'trailingEmptyStrict' or 'trailingEmpty'.

Non-positional sequence - Occurrences in the sequence cannot be identified by their position in the data alone. Often the components of such a sequence have an initiator. Such sequences sometimes allow the separator to be omitted for optional zero-length occurrences anywhere in the sequence. Speculative parsing might need to be employed by the parser to identify each occurrence. In DFDL, a sequence is non-positional if it contains any optional or array elements that have dfdl:occursCountKind 'parsed' or 'stopValue', and/or it has dfdl:separatorSuppressionPolicy 'anyEmpty'.

Property Name	Description
separator	List of DFDL String Literals or DFDL Expression Specifies a whitespace separated list of alternative DFDL String Literals that are the possible separators for the sequence. Separators occur in the data either before, between or after all occurrences of the elements or groups that are the children of the sequence, in accordance with dfdl:separatorPosition and dfdl:separatorSuppressionPolicy. Elements with dfdl:inputValueCalc have no representation in the data stream, and so never have an associated separator. This property can be computed by way of an expression which returns a string of whitespace separated values. The expression must not contain forward references to elements which have not yet been processed. It is a Schema Definition Error if the expression returns an empty string. This property can be used to determine the length of an element as described in Section 12.3.2 dfdl:lengthKind 'delimited'. Each string literal in the list, whether apparent in the schema, or returned as the value of an expression, is restricted to allow only certain kinds of DFDL String Literal syntax: • DFDL character entities are allowed. • DFDL Byte Value entities (%#rXX;) are allowed. • DFDL Character Class ES is not allowed. • DFDL Character Classes NL, WSP, WSP+, and WSP* are allowed. • The WSP* entity cannot appear on its own as one of the string literals in the list when determining the length of a component by scanning for delimiters. If the above rules are not followed it is a Schema Definition Error. The ***Separator, PrefixSeparator*** and ***PostfixSeparator*** regions contain one of the strings specified by the dfdl:separator property. When this

	property has "" (empty string) as its value then the separator region is of length zero. When parsing, the list of values is processed in a greedy manner, meaning it takes all the separators, that is, each of the string literals in the whitespace separated list, and matches them each against the data. The separator with the longest match is the one that is selected as having been 'found'. Once a matching separator is found, no other matches are subsequently attempted (i.e., there is no backtracking). On unparsing the first separator in the list is used as the separator. If a child element uses an escape scheme, then the escape scheme also applies to any separator; hence, if the separator appears within the element value, it is escaped. If dfdl:ignoreCase is 'yes' then the case of the string is ignored by the parser. Annotation: dfdl:sequence, dfdl:group (sequence)
separatorPosition	Enum Valid values 'infix', 'prefix', 'postfix' 'infix' means the separator occurs between the elements in the ***Separator*** grammar region. 'prefix' means the separator occurs before each element in the ***Separator*** grammar region and the ***PrefixSeparator*** grammar region. 'postfix' means the separator occurs after each element in the ***Separator*** grammar region and the ***PostfixSeparator*** grammar region. Annotation: dfdl:sequence, dfdl:group (sequence).
separatorSuppressionPolicy	Enum Valid values 'never', 'anyEmpty', 'trailingEmpty', 'trailingEmptyStrict' Only applicable if dfdl:separator is not "" (empty string) and dfdl:sequenceKind is 'ordered'. Controls the circumstances when separators are expected in the data when parsing, or generated when unparsing, if an optional element occurrence or a group has a zero-length representation. See Section 14.2.1 Separators and Suppression. When dfdl:sequenceKind is 'unordered' then 'anyEmpty' is implied. Annotation: dfdl:sequence, dfdl:group (sequence)

Table 45 Properties for Sequence Groups with Separators

14.2.1 Separators and Suppression

When parsing a sequence group that specifies a separator, the number of occurrences and separators that are expected in the data stream for a child (element or group) depends on several factors:

- Whether element occurrences are optional or required
- Whether the occurrences (element or group) have a zero-length representation
- Whether occurrences (element or group) are trailing
- Whether the sequence is positional
- The dfdl:occursCountKind of the element

Where to expect a separator for optional content of zero-length is controlled by property dfdl:separatorSuppressionPolicy.

separatorSuppressionPolicy	Implications
never	Positional sequence where all occurrences must be found in the data, along with their associated separator.
trailingEmptyStrict	Positional sequence where *trailing occurrences* that have zero length representation must be omitted from the data, along with their associated separator.

trailingEmpty	Positional sequence where *trailing occurrences* that have zero length representation may be omitted from the data, along with their associated separator.
anyEmpty	Non-positional sequence where any occurrences that have zero length representation may be omitted from the data, along with their associated separator. It must be possible for speculative parsing to identify which elements are present.

Table 46 Sequence groups and separator suppression

The following are definitions for terminology used in this section:

Potentially trailing element – An array or optional element describes an occurrence that is said to be *potentially trailing* if the element can have a zero length representation and is followed in its enclosing group definition by only these kinds of schema components:

1. calculated elements (those having dfdl:inputValueCalc)
2. additional potentially trailing elements
3. potentially trailing groups

Intuitively, the array or optional element occurrence could be last.

Potentially trailing group – A group is said to be *potentially trailing* if the group has no framing and contains only potentially trailing element declarations/references, or recursively similar sequence or choice groups, and is followed in its enclosing group definition by only additional potentially trailing elements or potentially trailing groups.

Trailing or Actually Trailing – An element occurrence or group occurrence in the data is said to be *actually trailing* if it is potentially trailing and has zero-length representation and is not followed in the data by any other non-zero length element occurrence or group occurrence limited by the end of the enclosing sequence group.

In the sections that follow, it is important to remember that the dfdl:separatorSuppressionPolicy property is carried on the sequence, while the XSD minOccurs, XSD maxOccurs and dfdl:occursCountKind properties are carried on an *element* in that sequence.

14.2.2 Parsing Sequence Groups with Separators

Parsing child elements is described first. Parsing for child groups is described in Section 14.2.2.3.

When an element is required and is not an array then one occurrence is always expected along with its separator. The dfdl:separatorSuppressionPolicy of the sequence has no effect (nothing is eligible for suppression). Otherwise the behaviour is dependent on dfdl:occursCountKind.

When dfdl:occursCountKind is 'fixed' then XSD minOccurs must equal maxOccurs and that many occurrences are always expected along with their separators. The dfdl:separatorSuppressionPolicy of the sequence has no effect (nothing is eligible for suppression).

When dfdl:occursCountKind is 'expression' the number of occurrences is given by dfdl:occursCount and exactly that many occurrences are always expected along with their separators. The dfdl:separatorSuppressionPolicy of the sequence has no effect (nothing is eligible for suppression).

When dfdl:occursCountKind is 'parsed' any number of occurrences and their separators are expected. The dfdl:separatorSuppressionPolicy of the sequence must be 'anyEmpty' and it is a Schema Definition Error otherwise.

When dfdl:occursCountKind is 'stopValue', any number of occurrences and their separators are expected followed by the stop value and its separator. The dfdl:separatorSuppressionPolicy of the sequence has no effect.

When dfdl:occursCountKind is 'implicit', between XSD minOccurs and XSD maxOccurs (inclusive) occurrences and their separators are expected, according to the dfdl:separatorSuppressionPolicy of the sequence.

The behaviour for 'implicit' is more fully expressed in matrix form. The cells in the matrix give the number of occurrences of element values that are expected in the data stream when parsing, for the different values of dfdl:separatorSuppressionPolicy. The number of occurrences also depends whether XSD maxOccurs is unbounded or not, and the position of the element in the sequence. The number of separators can be inferred from this, considering dfdl:separatorPosition.

	dfdl:occursCountKind 'implicit'	
	Potentially Trailing	**Not Potentially Trailing**

<table>
<tr><th rowspan="2">dfdl:
separator-
Suppression-Policy</th><th colspan="2">maxOccurs unbounded</th><th colspan="2">maxOccurs bounded</th><th rowspan="2">maxOccurs
unbounded</th><th rowspan="2">maxOccurs
bounded</th></tr>
<tr><th>Element not declared last</th><th>Element declared last</th><th>Element declared last or occurrence followed by end-of-group</th><th>Element not declared last and occurrence not followed by end-of-group</th></tr>
<tr><td>never</td><td colspan="2" rowspan="3">Schema definition error</td><td colspan="2" rowspan="2"></td><td rowspan="3">Schema definition error</td><td rowspan="2">RepDef(min) ~ Rep(max - min)</td></tr>
<tr><td>trailingEmptyStrict</td><td>RepDef(min) [~ Rep(M < INF) ~ RepNonZero(1)]</td><td>RepDef(min) [~ Rep(M < max - min) ~ RepNonZero(1)]</td><td rowspan="2">RepDef(min) ~ Rep(max - min)</td></tr>
<tr><td>trailingEmpty</td><td rowspan="2">RepDef(min) ~ Rep(M < INF)</td><td rowspan="2">RepDef(min) ~ Rep(M <= max - min)</td><td rowspan="2">RepDef(min) ~ Rep(M < INF)</td><td rowspan="2">RepDef(min) ~ Rep(M <= max - min)</td></tr>
<tr><td>anyEmpty</td><td colspan="2"></td></tr>
</table>

Table 47 Separator Suppression for dfdl:occursCountKind 'implicit' when Parsing

The notation in each cell uses the "~" symbol to mean "followed by" in the data stream. Square brackets surround things that are optional, as in they may or may not appear in the data stream.

The descriptions found in the cells of the matrix do not provide a parsing algorithm, but rather state declaratively a pattern that the data must match in order to be correctly parsed.

RepDef(min) is short for "representation" and "defaultable". It means XSD minOccurs occurrences of nil, empty or normal representation[45]. These are required occurrences, so default rules apply for empty representations. XSD minOccurs may be 0, in which case there are no required occurrences.

Rep(M) means M occurrences of nil, empty, normal or absent representation. These are optional occurrences, so default rules do not apply for empty representations.

RepNonZero(1) means an occurrence of a nil, empty or normal representation where such a representation does not have zero-length[46]. This is an optional occurrence, so default rules do not apply.

A notation like ***Rep(M <= max – min)*** means that there are M occurrences, where M is some value between the values of the XSD minOccurs and XSD maxOccurs properties. When an unbounded number of occurrences is possible this is shown explicitly by *Rep(M < INF)* , INF meaning infinity or unbounded.

14.2.2.1 Errors When the Sequence is Positional

In the matrix abovethere are some cells where the combination of properties doesn't make sense, and a Schema Definition Error is raised. These occur when an element has dfdl:occursCountKind 'implicit' and XSD maxOccurs 'unbounded', and dfdl:separatorSuppressionPolicy implies that the sequence is positional, specifically:

- If a sequence has dfdl:separatorSuppressionPolicy 'never';
- If a sequence has dfdl:separatorSuppressionPolicy 'trailingEmptyStrict' or 'trailingEmpty' and the element is not the last declaration in the sequence. (This avoids ambiguity about which element is being suppressed.)

14.2.2.2 Example Parsing Scenarios

Consider the cell of the matrix above for the element in this DFDL schema fragment:

```
<xs:sequence dfdl:separator='|' dfdl:separatorPosition='infix'
        dfdl:separatorSuppressionPolicy='trailingEmptyStrict'>
  <xs:element name='a' type='xs:int' default='0'
           maxOccurs='5' minOccurs='0'
           dfdl:representation='text' dfdl:textNumberPattern='#0'
           dfdl:occursCountKind='implicit'
           dfdl:initiator='[' dfdl:terminator=']'
           dfdl:emptyValueDelimiterPolicy='both' />
</xs:sequence>
```

[45] Absent representation implies Processing Error for ‘implicit’ when less than or equal to XSD minOccurs.

[46] Absent representation always implies zero-length. Nil, empty, and normal representations can also be zero-length with the right combinations of properties. See Section 9.2.5 Zero-length Representation.

Within the sequence this element 'a' is clearly potentially trailing as it is declared last. The corresponding cell in the matrix above contains this description:

RepDef(min) [~ Rep(M < max – min) ~ RepNonZero(1)]

Since XSD minOccurs='0', the first term, RepDef(min) vanishes, leaving:

Rep(M < max - min) ~ RepNonZero(1)

Note Rep(M) permits absent representations, and if encountered they are simply omitted from the Infoset.

So, this data

```
[1]|[2]|[3]|[4]|[5]
```

parses and 5 items appear in the Infoset.

This data

```
|||[4]
```

also parses because absent representations are accepted, but only one item appears in the Infoset.(The fact that the occurrence is fourth in the array is not preserved into the Infoset). However, this data

```
|||[4]|
```

causes a Processing Error because there is a final trailing separator and dfdl:separatorSuppressionPolicy is 'trailingEmptyStrict'.

Now consider the same scenario but XSD minOccurs of '2'. The first term reappears as *RepDef(2)*. The data

```
|||[4]
```

which previously parsed successfully would now cause a Processing Error because the first two occurrences are required, so they must be either a normal representation, that is, matching xs:int syntax with surrounding initiator and terminator, or the empty representation which is `[]`. An example which parses correctly with XSD minOccurs of '2' is:

```
[1]|[]||[4]
```

In this case the Infoset contains 3 items with values 1, 0, 4. The 0 value arises because the occurrence has the empty representation, the occurs index is 2 so it is required, and there is a default value 0.

If the scenario is changed so that dfdl:separatorSuppressionPolicy is 'trailingEmpty' then a different cell of the matrix above applies.

RepDef(min) [~ Rep(M < max – min)]

This has a more lax behavior so that this data is also acceptable:

```
[1]|[]||[4]|
```

In this case the final trailing separator is tolerated, though when unparsing this final trailing separator would not be created. This is a case where what is parsed is not exactly recreated on unparsing from the resulting Infoset, but all the information content is preserved.

Now consider the same scenario but XSD maxOccurs is 'unbounded'. In that case this data is acceptable:

```
[1]|[]||||||||||||||||||||||||||[4]||||||||||||||||||||||
```

The Infoset values are again 1, 0, 4. But all the excess separators are tolerated.

14.2.2.3 Parsing Child Groups within Separated Sequences

When a child of a sequence is a group then a separator is expected/tolerated depending on dfdl:separatorSuppressionPolicy and other factors:

- 'never' - the child group's associated separator is expected
- 'trailingEmpty' – if the child group is potentially trailing, has zero-length and it is actually trailing, its separator may appear or not. Additional separators are not expected.
- 'trailingEmptyStrict' – if the child group is potentially trailing, has zero-length and it is actually trailing, its separator must not appear.
- 'anyEmpty' – if the child group has zero-length its separator must not appear.

14.2.3 Unparsing Sequence Groups with Separators

Unparsing child elements is described first. Unparsing for child groups is described in Section 14.2.3.2.

When an element is required and is not an array then one occurrence is always output along with its separator. The dfdl:separatorSuppressionPolicy of the sequence has no effect (nothing is eligible for suppression).

Otherwise the behaviour is dependent on dfdl:occursCountKind.

When dfdl:occursCountKind is 'fixed' or 'expression' the occurrences in the augmented Infoset are always output along with their separators. The dfdl:separatorSuppressionPolicy of the sequence has no effect (nothing is eligible for suppression).

When dfdl:occursCountKind is 'parsed' non zero-length occurrences in the augmented Infoset are output along with their separators. The dfdl:separatorSuppressionPolicy of the sequence must be 'anyEmpty' and it is a Schema Definition Error otherwise.

When dfdl:occursCountKind is 'stopValue' the occurrences in the augmented Infoset are output along with their separators followed by the stop value and its separator, according to the dfdl:separatorSuppressionPolicy of the sequence.

When dfdl:occursCountKind is 'implicit' the occurrences in the augmented Infoset are output along with their separators, according to the dfdl:separatorSuppressionPolicy of the sequence.

The behaviour for 'implicit' is more fully expressed in matrix form. The cells in the matrix give the number of occurrences of element values that are output to the data stream when unparsing, for the different values of dfdl:separatorSuppressionPolicy. The number of occurrences also depends whether XSD maxOccurs is unbounded or not, and the position of the element in the sequence. The number of separators output can be inferred from this, considering dfdl:separatorPosition.

<table>
<tr><th rowspan="4">dfdl:
separatorSuppressionPolicy</th><th colspan="5">dfdl:occursCountKind 'implicit'</th></tr>
<tr><th colspan="4">Potentially Trailing</th><th colspan="2">Not Potentially Trailing</th></tr>
<tr><th colspan="2">maxOccurs unbounded</th><th colspan="2">maxOccurs bounded</th><th rowspan="2">maxOccurs unbounded</th><th rowspan="2">maxOccurs bounded</th></tr>
<tr><th>Element not declared last</th><th>Element declared last</th><th>Element declared last or occurrence followed by end-of-group</th><th>Element not declared last and occurrence not followed by end-of-group</th></tr>
<tr><td>never</td><td colspan="2" rowspan="3">Schema definition error</td><td colspan="2" rowspan="2">Unparse N occurrences ~ unparse (maxOccurs -- N) trailing zero-length occurrences</td><td rowspan="3">Schema definition error</td><td rowspan="3">Unparse N occurrences ~ unparse (maxOccurs -- N) trailing zero-length occurrences</td></tr>
<tr><td>trailingEmptyStrict</td></tr>
<tr><td>trailingEmpty</td><td colspan="2">Unparse N occurrences (suppressing trailing zero-length occurrences)</td></tr>
<tr><td>anyEmpty</td><td colspan="6">Unparse N occurrences (suppressing any optional zero-length occurrences)</td></tr>
</table>

Table 48 Separator Suppressions for dfdl:occursCountKind 'implicit'

The notation in each cell uses the "~" symbol to mean "followed by" in the output data stream.

N stands for the number of elements in the augmented Infoset, which includes any defaults.

unparse N occurrences means output N unparsed Infoset items and associated separators.

unparse(M) trailing zero length occurrences means output M adjacent separators (according to dfdl:separatorPosition) as if separating M element occurrences.

(suppressing trailing zero-length reps) implies the unparser MUST look ahead into the Infoset and determine when the representations are zero-length, and then identify those in trailing position. No separators are output corresponding to the trailing zero-length representations.

14.2.3.1 Example Unparsing Scenarios

Consider the cell of the matrix above for the element in this DFDL schema fragment:

```
<xs:sequence dfdl:separator='|' dfdl:separatorPosition='infix'
          dfdl:separatorSuppressionPolicy='trailingEmpty'>
  <xs:element name='a' type='xs:int'
           maxOccurs='5' minOccurs='0'
           nillable='true'
           dfdl:representation='text' dfdl:textNumberPattern='#0'
           dfdl:occursCountKind='implicit'
           dfdl:initiator='[' dfdl:terminator=']'
           dfdl:emptyValueDelimiterPolicy='none'
           dfdl:nilKind='literalValue' dfdl:nilValue='%ES;'
           dfdl:nilValueDelimiterPolicy='none' />
</xs:sequence>
```

This example is similar to the one used above in the discussion of parsing with separator suppression. However, the element has no default value, the dfdl:emptyValueDelimterPolicy has been removed, and the element is nillable. Element 'a' is clearly potentially trailing as it is declared last. The corresponding cell in the matrix above contains this description:

unparse N occurrences (suppressing trailing zero length reps)

Assume unparsing an Infoset containing five values: 1, 0, nil[47], 4, nil. Five occurrences are unparsed; however, the last value is nil, which has a representation of '%ES;' meaning empty-string, and dfdl:nilvalueDelimiterPolicy is 'none' meaning no initiator or terminator is to appear in the data. Since the schema is suppressing trailing zero-length reps the unparse results in this output:

```
[1]|[0]||[4]
```

This is an example where if the data is reparsed, it does not result in that original Infoset, because the trailing empty value which is the representation of the nil value, is not represented in the output, and so does not cause an Infoset item with **[nilled]** true to be created in the Infoset when this data is parsed. To preserve the nil, change the dfdl:nilValueDelimiterPolicy to 'both'. In that case the output would be:

```
[1]|[0]|[]|[4]|[]
```

The nils now have explicit representation in the data and are recreated in the Infoset when parsing.

14.2.3.2 Unparsing Child Groups within Separated Sequences

When a child of a sequence is a group then a separator is output depending on dfdl:separatorSuppressionPolicy and other factors:

- 'never' - the child group's associated separator is output
- 'trailingEmpty' or 'trailingEmptyStrict' – if the child group is potentially trailing, has zero-length and it is actually trailing, its separator is not output.
- 'anyEmpty' – if the child group has zero-length its separator is not output.

14.3 Unordered Sequence Groups

The occurrences of members of a sequence group with dfdl:sequenceKind of 'unordered' (hereafter referred to as an 'unordered sequence') may appear in the data in any order. Occurrences of the same member do not have to be contiguous. In the Infoset, sequence groups are always in schema order, so a DFDL processor MUST sort the members of an unordered sequence into schema order when parsing. When unparsing, the Infoset must already be in schema order, and the members of the sequence are output in schema order.

14.3.1 Restrictions for Unordered Sequences

It is a Schema Definition Error if any member of the unordered sequence is not an element declaration or an element reference.

It is a Schema Definition Error if a member of an unordered sequence is an optional element or an array element and its dfdl:occursCountKind property is not 'parsed'

It is a Schema Definition Error if two or more members of the unordered sequence have the same name and the same namespace (see post-processing transformation below)

It is a Schema Definition Error if an unordered sequence has no members.

14.3.2 Parsing an Unordered Sequence

When parsing, the semantics of an unordered sequence are expressed by way of:

1. a source-to-source transformation of the sequence group definition, and
2. a post-processing transformation of the Infoset .

An implementation MAY use any technique consistent with this semantic.

14.3.2.1 Source-to-source Transformation

The source-to-source transformation turns the declaration of an unordered sequence into an ordered sequence group that contains a repeating choice. To ensure that the resulting schema is a valid DFDL schema, the choice group is wrapped in an array element.

The unordered sequence is transformed as follows:

[47] An Infoset item value of nil means the Infoset item **[nilled]** member is true, and the **[dataValue]** member has no value. See Section 4.2.2 Element Information Items.

- the dfdl:sequenceKind property of the unordered sequence is changed to "ordered"
- the content of the unordered sequence is replaced by a complex element (the 'choice element') with the following properties:
  - XSD minOccurs="0"
  - XSD maxOccurs="unbounded"
  - dfdl:lengthKind "implicit"
  - dfd:occursCountKind "parsed"
- the content of the choice element's complex type is a choice group with the following properties:
- dfdl:choiceLengthKind "implicit"
- The members of the unordered sequence become the members of the choice group, with their declaration order preserved.
- The XSD minOccurs and XSD maxOccurs properties on each member of the choice group are both set to 1.

Using the following example as an illustration:

```
<xs:sequence dfdl:sequenceKind="unordered" dfdl:separator=",">
   <xs:element name="a" type="xs:string"  dfdl:initiator="A:" />
   <xs:element name="b" type="xs:int" minOccurs="0" dfdl:initiator="B:" />
   <xs:element name="c" type="xs:string" minOccurs="0" maxOccurs="10"
     dfdl:initiator="C:" />
</xs:sequence>
```

The above unordered sequence group is conceptually rewritten into the following ordered sequence group:

```
<xs:sequence dfdl:sequenceKind="ordered" dfdl:separator=",">
  <xs:element name="choiceElement" minOccurs="0" maxOccurs="unbounded"
                  dfdl:occursCountKind="parsed">
    <xs:complexType>
      <xs:choice dfdl:choiceLengthKind="implicit">
        <xs:element name="a" type="xs:string" dfdl:initiator="A:" />
        <xs:element name="b" type="xs:int" dfdl:initiator="B:" />
        <xs:element name="c" type="xs:string" dfdl:initiator="C:" />
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:sequence>
```

Processing then constructs a temporary info set for this ordered sequence group by parsing the data.

If a member element is found to have the empty representation then the parsing of that element must use the original value of XSD minOccurs. In this example, element "b" has XSD minOccurs "0" and if it is found with the empty representation then it must not be defaulted.

14.3.2.2 **Post-processing Transformation**

Post-processing consists of the following steps:

1. Sort the temporary Infoset to produce the real Infoset
2. Check scalar elements and validate

Step 1: Sort the Temporary Infoset

The temporary Infoset is transformed into the Infoset conforming to the original unordered sequence. All members of the temporary Infoset having the same name and namespace as the first child of the unordered sequence are placed first, in the order in which they were parsed. This algorithm repeats for the second child of the unordered sequence and so on until all members of the temporary Infoset have been sorted into the schema declaration order of the original unordered sequence.

For the example above, the temporary Infoset is transformed into the Infoset corresponding to:

```
<xs:sequence>
  <xs:element name="a" type="xs:string" />
  <xs:element name="b" type="xs:int" minOccurs="0" />
  <xs:element name="c" type="xs:string" minOccurs="0" maxOccurs="10" />
</xs:sequence>
```

Step 2: Check Scalar Elements and Validate

For each element in the unordered sequence having XSD minOccurs "1" and maxOccurs "1", the number of occurrences is checked. Each such element must occur exactly once in the Infoset, else it is a Processing Error.

If validation is enabled, the DFDL processor validates the number of occurrences of each member of the unordered sequence against XSD minOccurs and XSD maxOccurs.

These checks are the same as those performed for an ordered sequence group. However, in an unordered sequence the checking of XSD minOccurs and XSD maxOccurs MUST be performed after the entire group has been parsed.

14.3.3 Unparsing an Unordered Sequence

When unparsing, the behavior is exactly as if dfdl:sequenceKind is 'ordered'. The members of the unordered sequence group are output in schema declaration order.

14.4 Floating Elements

Elements within an ordered sequence can be designated as floating which means that they can appear in any position within the sequence.[48]

An ordered sequence with floating components is similar to an unordered sequence except only the floating elements may be out of order.

Within an ordered sequence with floating components a non-floating array element must have its occurrences appearing contiguously, so any floating element occurrences cannot appear in between occurrences of the array element.(In other words, property dfdl:floating 'yes' only makes a statement about the floating element, not about any other elements in the sequence.)

Property Name	Description
floating	Enum Valid values are 'yes', 'no' Whether the occurrences of an element in an ordered sequence can appear out-of-order in the representation. When parsing, and dfdl:floating is 'yes', occurrences of the element may be encountered in the representation in many positions within its containing sequence. If present they are placed into the Infoset in schema declaration order. If the element repeats, occurrences do not need to be contiguous in the representation. When parsing, and dfdl:floating is 'no', occurrences of the element must be in schema declaration order, and, if present, they are placed into the Infoset in schema declaration order. It is a Processing Error if instances of the element are not encountered in schema declaration order. When unparsing, occurrences of the element are expected in the Infoset in schema declaration order and are output in the representation in schema declaration order. It is a Processing Error if occurrences of the element are not encountered in schema declaration order, It is a Schema Definition Error if an unordered sequence or a choice contains any element with dfdl:floating 'yes'. It is a Schema Definition Error if an ordered sequence contains any element with dfdl:floating 'yes' and also contains non-element component (such as a choice or sequence model group). It is a Schema Definition Error if an element with dfdl:floating 'yes' is an optional element or an array element and its dfdl:occursCountKind property is not 'parsed' It is a Schema Definition Error if two or more elements with dfdl:floating 'yes' in the same group have the same name and the same namespace. Annotation: dfdl:element

Table 49 Properties for Floating Elements

An ordered sequence of N element children with dfdl:floating 'yes' is equivalent to an unordered sequence with the same N element children with dfdl:floating 'no'.

[48]The NTE segment in the X12 EDI standard is an example of a floating element.

A complex element with dfdl:floating 'yes' can have as its content model a sequence with elements that also have dfdl:floating 'yes'.

Every element in a sequence containing one or more floating elements is a point of uncertainty, similar to the way every element in an unordered sequence is a point of uncertainty.

In resolving this point of uncertainty, a parser MUST look for the element defined at that position in the schema first and only if unsuccessful with parsing that element, the parser MUST subsequently attempt to parse the floating elements in the order they are defined in the schema. As soon as any such parse is successful this resolves the point of uncertainty.

14.5 Hidden Groups

Some fields in the physical stream provide information about other fields in the stream and are not really part of the data. For example, a field can give the number of repeats in a following array. These fields may not be of interest to an application after the data has been parsed, and so may be removed from the Infoset on parsing by containing the element declarations for them within a hidden group. A hidden group allows elements to be defined that are not added to the Infoset on parsing and are not expected in the Infoset on unparsing.

```
<xs:element name="root">
  <xs:complexType>
    <xs:sequence>

      <xs:sequence>
        <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:sequence hiddenGroupRef="tns:hiddenRepeatCount">
        </xs:appinfo></xs:annotation>
      </xs:sequence>

      <xs:element name="arrayElement" type="xs:int"
                  minOccurs="0" maxOccurs="unbounded"
                  dfdl:occursCountKind="expression"
                  dfdl:occurCount= "{../repeatCount}"
                  dfdl:representation="binary" dfdl:lengthKind="implicit" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:group name="hiddenRepeatCount" >
  <xs:sequence>
    <xs:element name="repeatCount" type="xs:int"
                dfdl:outputValueCalc="{count(../arrayElement)}"
                dfdl:representation="binary" dfdl:lengthKind="implicit" />
  </xs:sequence>
</xs:group>
```

An element contained within the extent of a hidden group is commonly called a hidden element.

Hidden elements are referenced via path expressions using the same DFDL expression that would be used if they were not hidden.

Hidden elements can (typically will) contain the regular DFDL annotations to define their physical properties and on unparsing to set their value. They are processed using the same behavior as non-hidden elements.

When the dfdl:hiddenGroupRef property is specified on an xs:sequence schema component, the appearance of any other DFDL properties on that component is a Schema Definition Error. It is also a Schema Definition Error if the sequence is not empty.

It is a Schema Definition Error if the sequence is the only thing in the content model of a complex type definition.

It is a Schema Definition Error if dfdl:hiddenGroupRef appears on a xs:group reference, that is, unlike most format properties that apply to sequences, dfdl:hiddenGroupRef cannot be combined from a xs:group reference.

A hidden group may appear within another hidden group.

Property Name	Description
hiddenGroupRef	QName

	Reference to a global model group definition. Elements within this model group are not added to the Infoset and are called hidden elements. The model group within the model group definition may be a xs:sequence or xs:choice It is a Schema Definition Error if the value is the empty string. It is not possible to place this property in scope on a dfdl:format annotation. Annotation: dfdl:sequence

Table 50 Properties for Hidden Groups

When unparsing a hidden group, the behaviour is the same as when elements are missing from the Infoset; that is, the default-values algorithm applies. The only difference is that if a required element does not have a default value or a dfdl:outputValueCalc then it is a Schema Definition Error instead of a Processing Error. Note that this can be checked statically.

When unparsing a hidden group, it is a Processing Error if an element information item is provided in the Infoset for a hidden element.

Examples of hidden groups are in Section 17 Calculated Value Properties.

15 Choice Groups

A choice corresponds to concepts variously called variant records, multi-format records, discriminated unions, or tagged unions in various programming languages. In some contexts, choices are referred to generally as 'unions'. However, this should not be confused with XSD unions which are an unrelated concept.

The following properties are specific to xs:choice.

Property Name	Description
choiceLengthKind	Enum Valid values are 'implicit', 'explicit' 'implicit' means the branches of the choice are not filled, so the ChoiceContent region is variable length depending on which branch appears. 'explicit' means that the branches of the choice are always filled to the fixed length specified by dfdl:choiceLength, so the ChoiceContent region is fixed-length regardless of which branch appears. Annotation: dfdl:choice, dfdl:group (choice)
choiceLength	Integer Only used when dfdl:choiceLengthKind is 'explicit'. Specifies the length of the choice in bytes, so the ChoiceContent region is fixed-length regardless of which branch appears. A ***ChoiceUnused*** region is therefore possible which when unparsing is filled with dfdl:fillByte. Annotation: dfdl:choice, dfdl:group (choice)
initiatedContent	Enum Valid values are 'yes', 'no' When 'yes' indicates that all the branches of the choice are initiated. It is a Schema Definition Error if any children have their dfdl:initiator property set to the empty string. The branch is known to exist when its initiator has been found. Any subsequent error parsing the branch does not cause the parser to backtrack. When 'no', the branches of the choice may have their dfdl:initiator property set to the empty string. Annotation: dfdl:sequence, dfdl:choice, dfdl:group
choiceDispatchKey	DFDL Expression The expression must evaluate to an xs:string. It is a Schema Definition Error if the expression returns an empty string. It is a Schema Definition Error if the expression contains forward references to elements which have not yet been processed. This property is used only when parsing. The resultant string must match one of the dfdl:choiceBranchKey property values of one of the branches of the choice. This match is *case sensitive*. If so, it discriminates to that branch. The parser then goes straight to that branch, ignoring consideration of any other choice branches. No backtracking of this decision occurs if there is a subsequent Processing Error. It is a Processing Error if the value of the expression does not match any of the dfdl:choiceBranchKey property values for any of the branches. It is a Schema Definition Error if any choice branch does not specify a dfdl:choiceBranchKey in a choice that carries choiceDispatchKey. It is not possible to place this property in scope on a dfdl:format annotation. Annotation: dfdl:choice
choiceBranchKey	List of DFDL String Literals This literal provides an alternate way to discriminate a choice to a branch. When the dfdl:choiceDispatchKey expression evaluates to a string matching one of this property's values, the choice is discriminated to this branch. The match is case sensitive.

	It is a Schema Definition Error if individual dfdl:choiceBranchKey values are not unique across all branches of a choice that carries dfdl:choiceDispatchKey. It is a Schema Definition Error if dfdl:choiceBranchKey is specified on a global element, or on a sequence or choice that is the child of a global group definition. It is a Schema Definition Error if any choice branch does not specify a dfdl:choiceBranchKey in a choice that carries choiceDispatchKey. Byte value entities are not allowed. Character classes are not allowed. This property is only used when parsing. It is not possible to place this property in scope on a dfdl:format annotation. Annotation: dfdl:element, dfdl:sequence, dfdl:choice, dfdl:group

Table 51 Properties for Choice Groups

A choice can have a dfdl: initiator and/or a dfdl:terminator as described earlier.

The explanation of choices requires this terminology:

Branch - A *branch* is one of the available alternatives within a choice. A branch can be an element of simple type or complex type, or it can be an embedded sequence, choice or group reference.

Root of the Branch - Each branch conceptually has a single schema component at its root which is an element, sequence, choice or group reference. This component is known as the Root of the Branch

The Root of the Branch must not be optional. That is XSD minOccurs must be greater than zero.

A choice that declares no branches in the DFDL schema is a Schema Definition Error.

When processing a choice group, the parser validates any contained path expressions. If a path expression contained inside a choice branch refers to any other branch of the choice, then it is a Schema Definition Error. Note that this rule handles nested choices also. A path that navigates outward from an inner choice to another alternative of an outer choice is violating this rule with respect to the outer choice.

15.1 Resolving Choices

When processing a choice, there are two ways to resolve the intended branch. In one, speculative parsing is used. In the other, a constant-time direct dispatch to a branch is performed.

15.1.1 Resolving Choices via Speculation

Speculative resolution works as follows:

1. Attempt to parse the first branch of the choice.
2. If this fails with a Processing Error
   a) If a dfdl:discriminator evaluated to true earlier on this branch
      then the parser is 'bound' to this branch and parsing of the entire choice construct fails with a Processing Error.
   b) If the branch has a dfdl:initiator and the choice has dfdl:initiatedContent ‘yes’
      then the parser is 'bound' to this branch and parsing of the entire choice construct fails with a Processing Error.
   c) Otherwiserepeat from step 1 for the next branch of the choice.
3. It is a Processing Error if the branches of the choice are exhausted.
4. If a branch is successfully parsed without error, then that branch's Infoset becomes the Infoset for the parse of the choice construct.
5. If the branch is an element declaration having dfdl:occursCountKind='expression' or dfdl:occursCountKind='parsed', then zero instances are possible. If the branch parses successfully without a discriminator but produces no element occurrences, then the branch is considered missing, and the parser looks for the next branch

It is not possible for variable settings to be communicated from the speculative attempt to parse a branch to any other parsing situation. The speculative effort is completely isolated. Whether it succeeds or fails, neither the parse position in the source data, nor anything in the variable memory, nor the Infoset is affected.

Nested choices can require unbounded[49] look ahead into the data.

15.1.2 Resolving Choices via Direct Dispatch

Direct dispatch provides a constant-time dispatch to a choice branch independent of how many choice branches there are.

Direct dispatch is indicated by the dfdl:choiceDispatchKey property. This expression is evaluated to compute the string matching (case sensitive) one of the dfdl:choiceBranchKey property values of one of the choice branches.

When a match is found, it is as if a dfdl:discriminator had evaluated to true on that branch. It is selected as resolution of the choice, and there is no backtracking to try other alternative selections if a Processing Error occurs.

The dfdl:choiceBranchKey property can be placed on element references, local element declarations, local sequences, local choices, or group references. All values of dfdl:choiceBranchKey properties must be unique across all branches of a choice that carries a dfdl:choiceDispatchKey property and it is a Schema Definition Error otherwise.

Note that it is a Schema Definition Error if both dfdl:initiatedContent and dfdl:choiceDispatchKey are provided on the same choice. However, it is not an error if a discriminator exists on a choice branch along with a dfdl:choiceBranchKey.

15.1.3 Unparsing Choices

On unparsing there is the question of how one identifies the appropriate schema choice branch corresponding to the data in the Infoset. This is complicated by the fact that the children may not be elements. They may themselves be sequences or choices. The selection of the choice branch is as follows: The element in the Infoset is used to search the choice branches in the schema, in schema definition order, but without looking inside any complex elements. If the element occurs in a branch, then that branch is selected and if subsequently a Processing Error occurs, this selection is not revisited (that is, there is no backtracking). If the next element to unparse does not identify any branch of the choice, or there is no next element to unparse, then there must be a choice branch with no required elements and the first such branch would be selected for unparsing. A choice branch can consist only of a nest of model groups with no actual element content or only optional element content.

To avoid any unintended behavior, all the children of a choice can be modeled as elements.

15.1.3.1 Unparsing Choices in Hidden Groups

When a choice appears inside a hidden group, there are no corresponding Infoset elements as there are none for hidden groups. The first branch of the choice is unparsed. All elements contained in the branch must have default values or must have dfdl:outputValueCalc properties to compute their values, and it is a Schema Definition Error otherwise.

[49] Because DFDL v1.0 does not allow recursive definitions, the notion of unbounded here is limited by the depth of the DFDL schema, so is not truly unbounded as it would be if recursion were allowed.

16 Properties for Array Elements and Optional Elements

These properties are for array elements (XSD maxOccurs >1 or unbounded) or optional elements (XSD minOccurs 0 and XSD maxOccurs 1). The properties handle a logical one-dimensional array of any simple or complex type.

Property Name	Description
occursCountKind	Enum Specifies how the actual number of occurrences is to be established. Valid values 'fixed', 'expression', 'parsed', 'implicit' ,'stopValue'. 'fixed' means use the XSD maxOccurs property. 'expression' means use the dfdl:occursCount property. 'parsed' means that the number of occurrences is determined solely by speculative parsing. 'implicit' means that the number of occurrences is determined by speculative parsing in conjunction with the XSD minOccurs and XSD maxOccurs properties. 'stopValue' means look for a mandatory logical stop value which signifies the end of the occurrences. These values are described in detail in Section 16.1. Annotation: dfdl:element
occursCount	DFDL Expression Specifies the number of occurrences of the element. Required only when dfdl:occursCountKind is 'expression'. This property is computed by way of an expression which returns a non-negative integer. The expression must not contain forward references to elements which have not yet been processed. Annotation: dfdl:element,
occursStopValue	List of DFDL Logical Values A whitespace separated list of logical values that specify the alternative logical stop values for the element. Required only when dfdl:occursCountKind is 'stopValue'. When parsing then if an occurrence of the element has a logical value that matches one of the values in this list then the parser MUST not expect any more occurrences of the element. On unparsing the first value is inserted as an additional final occurrence in the array after all the occurrences in the Infoset have been output. The dfdl:occursStopValue property must not be empty string. Annotation: dfdl:element

Table 52 Properties for Array Elements and Optional Elements

When XSD minOccurs 1 and XSD maxOccurs 1, the above properties are not used, and the behavior is as if dfdl:occursCountKind is 'fixed' as described in Section 16.1.1.

16.1 The dfdl:occursCountKind property

16.1.1 dfdl:occursCountKind 'fixed'

The enum 'fixed' should be used when the number of occurrences is always the same. The number is provided by the XSD maxOccurs property.

When parsing, maxOccurs occurrences are expected in the data. It is a Processing Error if less than maxOccurs occurrences are found or defaulted. The parser stops looking for occurrences when maxOccurs have been found or defaulted. When maxOccurs is 0, no occurrences are looked for in the data.

When unparsing, maxOccurs occurrences are expected in the Infoset. It is a Processing Error if less than maxOccurs occurrences are found or defaulted. The processor stops looking for more occurrence in the

Infoset after maxOccurs occurrences have been found. When maxOccurs is 0, no occurrences are looked for in the Infoset or written.

It is a Schema Definition Error if XSD minOccurs is not equal to XSD maxOccurs.

16.1.2 dfdl:occursCountKind 'implicit'

The enum 'implicit' should be used when the number of occurrences is to be established using speculative parsing, and there are lower and upper bounds to control the speculation. The bounds are provided by the XSD minOccurs and XSD maxOccurs properties.

When parsing, up to maxOccurs occurrences are expected in the data. It is a Processing Error if less than XSD minOccurs occurrences are found or defaulted. The parser stops looking for occurrences when either XSD minOccurs have been found or defaulted and speculative parsing does not find another occurrence, or XSD maxOccurs have been found or defaulted. When XSD maxOccurs is 0, no occurrences are looked for in the data.

When unparsing, up to XSD maxOccurs occurrences are expected in the Infoset. It is a Processing Error if less than XSD minOccurs occurrences are found or defaulted. The processor stops looking for more occurrences in the Infoset after XSD maxOccurs occurrences have been found. When XSD maxOccurs is 0, no occurrences are looked for in the Infoset or written.

16.1.3 dfdl:occursCountKind 'parsed'

The enum 'parsed' should be used when the number of occurrences is to be established solely using speculative parsing.

When parsing, any number of occurrences is expected in the data. The parser stops looking for occurrences when speculative parsing does not find another occurrence. If validation is enabled, it is a Validation Error if less than XSD minOccurs occurrences are found or defaulted, or greater than XSD maxOccurs occurrences are found.

When unparsing, any number of occurrences is expected in the Infoset. If validation is enabled, it is a Validation Error if less than XSD minOccurs occurrences are found or defaulted, or if more than XSD maxOccurs occurrences are found.

16.1.4 dfdl:occursCountKind 'expression'

The enum 'expression' should be used when the number of occurrences is calculated by evaluating a DFDL expression.

When parsing, the dfdl:occursCount expression is evaluated and provides the number of occurrences expected in the data. It is a Processing Error if less than dfdl:occursCount occurrences are found or defaulted. The parser stops looking for occurrences when dfdl:occursCount occurrences have been found. If validation is enabled, it is a Validation Error if less than XSD minOccurs occurrences are found or defaulted, or more than XSD maxOccurs occurrences are found. When dfdl:occursCount is 0, no occurrences are looked for in the data.

When unparsing, any number of occurrences are expected in the Infoset. If validation is enabled, it is a Validation Error if less than XSD minOccurs occurrences are found or defaulted, or if more than XSD maxOccurs occurrences are found. The dfdl:occurs expression is not evaluated. The ‘count’ is the number of occurrences in the augmented Infoset.

It is a Schema Definition Error if dfdl:occursCount is not provided or in scope.

16.1.5 dfdl:occursCountKind 'stopValue'

The enum 'stopValue' should be used when the end of the array is signaled by an occurrence having a logical value that is equal to one of the specified 'stop values'.

When parsing, any number of occurrences is expected in the data, followed by an occurrence which is a stop value as specified by dfdl:occursStopValue. It is a Processing Error if a stop value occurrence is not found in the data (including when there are zero other occurrences). The parser stops looking for occurrences once a stop value has been found. If validation is enabled, it is a Validation Error if less than XSD minOccurs occurrences are found or defaulted, or more than XSD maxOccurs occurrences are found, not including the stop value.

When unparsing, the behavior is the same as for 'parsed', with the addition that a stop value occurrence is output after the last Infoset occurrence. If dfdl:occursStopValue provides multiple stop values then the first is used.

The stop value itself is **not** added to the Infoset when parsing. It is a Processing Error if a stop value is found in the Infoset when unparsing. (This ensures that the array can be reparsed, as the stop value is placed automatically and only at the end.)

It is a Schema Definition Error if dfdl:occursStopValue is not provided or in scope.

It is a Schema Definition Error if the type of the element is complex.

It is a Schema Definition Error if any of the stop values provided by dfdl:occursStopValue do not conform to the simple type of the element.

16.2 Default Values for Arrays

When parsing, required occurrences that have empty representation may trigger the application of a default value, as described in Section 9.4.2 Element Defaults When Parsing.

When unparsing, required occurrences that are missing from the Infoset may trigger the application of a default value, as described in Section 9.4.3 Element Defaults When Unparsing.

The application of default values is **not** dependent on dfdl:occursCountKind, only on whether the occurrence is required or optional, whether there is a default value specified, and whether the data contains the empty representation (parsing) or is missing (unparsing). Section 9.4 Element Defaults contains the details.

16.3 Arrays with DFDL Expressions

If the value of a DFDL property of an array element (other than dfdl:occursCount) is given by a DFDL Expression, then the expression MUST be re-evaluated for each occurrence of the element in case the value changes.

16.4 Points of Uncertainty

Arrays can have points of uncertainty depending on the value of dfdl:occursCountKind. See Section 9.3.3 Resolving Points of Uncertainty for details.

16.5 Arrays and Sequences

In some situations, arrays of elements and sequence groups of elements seem to be similar; however, there is no notion of the array itself independent of its contained elements. Arrays are distinctly different from sequence groups in this way.

A sequence can have its own initiator, and a complex element having that sequence as its content can also have its own initiator, so one can express two different initiators.

Unlike a sequence group, an array does not have its own initiator, terminator, or alignment. Those properties apply to each element occurrence of the array. To give an alignment, initiator, separator, or terminator to an entire array one must enclose the element declaration for the array in a sequence group and specify the alignment, separator, initiator, and terminator on the sequence group.

16.6 Forward Progress Requirement

An array is potentially unbounded if any of the following are true:

- dfdl:occursCountKind is 'stopValue'
- dfdl:occursCountKind is 'parsed'
- dfdl:occursCountKind is 'implicit' and XSD maxOccurs is unbounded

To prevent an infinite loop, the parsing of an array that is potentially unbounded MUST terminate when the parsing of an occurrence makes no forward progress. This is detected when the following are true:

- The occurrence is a point of uncertainty;
- The position in the data does not move during the parsing of the occurrence (including any associated Separator, PrefixSeparator or PostfixSeparator region);
- The occurrence is known-to-exist with empty representation or nil representation.

In this situation, no forward progress occurs, and no way of ever detecting the end of the array is possible. Upon termination of the array, any Infoset items already added to the array are retained except when dfdl:occursCountKind is 'stopValue' in which case this results in a Processing Error because the stop value will never be encountered.

Further, to prevent unnecessary consumption of resources for large bounded values of XSD maxOccurs, the parsing of an array must similarly terminate when the following are true:

- dfdl:occursCountKind is 'implicit';
- The occurrence is a point of uncertainty;
- The position in the data does not move during the parsing of the occurrence (including any associated Separator, PrefixSeparator, or PostfixSeparator region);
- The occurrence is known to exist with empty representation.

In this situation no forward progress occurs, and nothing is being added to the infoset. Note that this differs from the above array termination because nil representation does not cause detection of this lack of forward progress as nilled element items are added to the Infoset, and the array eventually terminates when it contains XSD maxOccurs occurrences.

16.7 Parsing Occurrences with Non-Normal Representation

When parsing a single array, it is possible to extract occurrences that have different representations (nil, empty, normal, absent) although with some values of dfdl:lengthKind certain combinations of representations are not possible.

Occurrences with nil representation are added to the Infoset with **[nilled]** member true.

Occurrences with empty representation may or may not be added to the Infoset, as described in Section 9.4. If a required occurrence is not added to the Infoset, it may be a Processing Error, dependent on dfdl:occursCountKind as described in Section 16.1.

Occurrences with absent representation are not added to the Infoset. For a required occurrence it may be a Processing Error, dependent on dfdl:occursCountKind as described in Section 16.1.

16.8 Sparse Arrays

Consider parsing an array where optional occurrences with empty representation are present in the data, but there are also later optional occurrences present with normal representation. Such an array is called a 'sparse array'.

If the indices of the occurrences are significant and need to be preserved, then the array may be modelled using an element with XSD nillable 'true', dfdl:nilKind 'literalValue', dfdl:nilValue '%ES;' and dfdl:nilValueDelimiterPolicy the same as dfdl:emptyValueDelimiterPolicy. The occurrences with empty representation now become occurrences with nil representation, and produce nil values in the Infoset, so the absolute positions of all occurrences are preserved.

If the indices of the occurrences are not significant, then the array should be modelled using an element with XSD nillable 'false'. Optional occurrences with empty representation do not create items in the Infoset, so the absolute positions of any optional occurrences with normal representation are not preserved. Optional occurrences with empty representation are therefore skipped.

17 Calculated Value Properties

This section describes properties which allow the creation of calculated elements. When parsing, the value of a calculated element is derived using a DFDL Expression, and not by processing bytes from the data stream. When unparsing, the value of a calculated element is derived using a DFDL Expression and is not obtained from the Infoset in the usual way.

Calculated elements allow a technique that is commonly called layering. In this technique, some elements are said to be in the physical layer, and some in the logical layer. When parsing, the logical layer values are computed from physical layer values. When unparsing the opposite occurs, that is the physical layer values are computed from the logical layer values.

Calculated elements are commonly used with hidden elements to hide the physical layer elements so that they do not become part of the Infoset.

When a DFDL Schema is used to both parse and unparse data, then a calculated element on parsing normally implies use of one or more calculated elements on unparsing.

These properties apply to elements of simple type.

Property Name	Description
inputValueCalc	DFDL Expression An expression that calculates the value of the element when parsing. It is a Schema Definition Error if the result type of the expression does not conform to the base type of the element. The element value created using dfdl:inputValueCalc is validated like any other element value (when validation is enabled). An element that specifies a dfdl:inputValueCalc expression has no representation of its own in the data stream. All other DFDL representation properties are ignored. When an element which carries this property appears in a sequence that has a separator, no separator is associated with the element. When parsing, no separator is expected in the input data. When unparsing, no separator is written to the output data. The element must not be optional nor an array nor be global. The DFDL Expression must not refer to this element nor cause a circular reference to this element. The expression must not contain forward references to elements which have not yet been processed. It is a Schema Definition Error if this property is specified on an element which has an XSD fixed or default property. It is a Schema Definition Error if dfdl:inputValueCalc and dfdl:outputValueCalc are specified on the same element. It is not possible to place this property in scope on a dfdl:format annotation. If this property appears on an element declaration or element reference schema component, the appearance of any other DFDL properties on that component is a Schema Definition Error. If this property appears on an element reference, then DFDL properties expressed on the referenced global element declaration or its type are ignored. If this property appears on an element declaration, then DFDL properties expressed on its type are ignored. Annotation: dfdl:element
outputValueCalc	DFDL Expression An expression that calculates the value of the current element when unparsing. The element must not be optional nor an array nor be global. It is a Schema Definition Error if the result type of the expression does not conform to the base type of the element. The value created using dfdl:outputValueCalc is validated like any other element value (when validation is enabled). The value for the element, if any, in the Infoset is ignored.

	The DFDL expression must not refer to this element nor cause a circular reference to this element. The expression may contain forward references to elements which have not yet been processed. It is a Schema Definition Error if dfdl:outputValueCalc is specified on an element which has an XSD fixed or default property. It is a Schema Definition Error if dfdl:inputValueCalc and dfdl:outputValueCalc are specified on the same element. It is not possible to place this property in scope on a dfdl:format annotation. Annotation: dfdl:element

Table 53 Calculated Value Properties

17.1 Example: 2d Nested Array

Consider this simple example. The data stream contains two elements giving the number of rows and number of columns of an array of numbers. The representation of the array is stored after these two elements.

```
<xs:complexType name="array">
  <xs:sequence dfdl:initiator="" >

    <xs:sequence dfdl:hiddenGroupRef="tns:hiddenArrayCounts"/>

    <xs:element name="rows" maxOccurs="unbounded"
                dfdl:occursCountKind="expression"
                dfdl:occursCount="{ ../nrows }">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="cols" type="xs:float" maxOccurs="unbounded"
                      dfdl:occursCountKind="expression"
                      dfdl:occursCount=" { ../../ncols } " />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:group name="hiddenArrayCounts" >
  <xs:sequence>
    <xs:element name="nrows" type="xs:unsignedInt"
                dfdl:representation="binary"
                dfdl:lengthKind="implicit"
                dfdl:outputValueCalc="{ count(../rows) }"/>
    <xs:element name="ncols" type="xs:unsignedInt"
                dfdl:representation="binary"
                dfdl:lengthKind="implicit"
                dfdl:outputValueCalc=
                "{ if ( count(../rows) ge 1 )
                   then
                      count(../rows[1]/cols)
                   else
                      0
                 }"/>
  </xs:sequence>
</xs:group>
```


In the example abovethere are two hidden elements named 'nrows' and 'ncols'. These hidden elements' values are computed when unparsing from the number of occurrences in the 'rows' and 'cols' repeating elements. The 'rows' and 'cols' repeating elements number of occurrences are computed when parsing from the hidden elements 'nrows' and 'ncols'.

17.2 Example: Three-Byte Date

Logically, the data is a date.

```
<xs:element name="d" type="date"/>
```

Physically, it is stored as 3 single byte integers.

The format of this data is expressed as this schema:

```
<xs:sequence dfdl:representation="binary">
              <xs:element name="mm" type="byte" />
              <xs:element name="dd" type="byte" />
              <xs:element name="yy" type="byte"/>
</xs:sequence>
```

This physical representation can be hidden so that it does not become part of the Infoset:

```
<xs:sequence>
  <xs:sequence dfdl:hiddenGroupRef="tns:hiddenpDate"/>

  <xs:element name="d" type="date">
    …
  </xs:element>
</xs:sequence>

<xs:group name="hiddenpDate" >
  <xs:sequence>
    <xs:element name="pdate">
      <xs:complexType>
        <xs:sequence dfdl:representation="binary">
          <xs:element name="mm" type="byte" />
          <xs:element name="dd" type="byte" />
          <xs:element name="yy" type="byte"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>
```

A calculation can be used to compute the logical date element 'd' from the physical 'pdate' when parsing:

```
<xs:sequence>
  ... hidden pdate here ...

  <xs:element name="d" type="date">
    <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:element>
        <dfdl:property name="inputValueCalc">
          {
           fn:date(fn:concat(if(../pdate/yy gt 50 )then "19" else "20",
                             if ( ../pdate/yy gt 9 )
                                then xs:string(../pdate/yy)
                                else fn:concat("0",
                                               xs:string(../pdate/yy)),
                                               "-",
                                               xs:string(../pdate/mm),
                                               "-",
                                               xs:string(../pdate/dd)))
          }
        </dfdl:property>
      </dfdl:element>
    </xs:appinfo></xs:annotation>
  </xs:element>
  ...
</xs:sequence>
```

The expression above assembles a string resembling, for example, "2005-12-17" or "1957-3-9" which is the string representation of a date that is acceptable to the `fn:date` constructor function. The hidden element `'pdate'` is referenced by relative paths. The expression `'../pdate/yy'` accesses an element of type `'int'`, and the `xs:string` constructor function turns it into an integer.

Finally,one must handle the unparse case where the physical layer is computed from the logical layer:

```
<xs:sequence dfdl:representation="binary"
  <xs:element name="mm" type="byte"
       dfdl:outputValueCalc="{ fn:month-from-date(../d) }" />
  <xs:element name="dd" type="byte"
       dfdl:outputValueCalc="{ fn:day-from-date(../d) }" />
  <xs:element name="yy" type="byte"
       dfdl:outputValueCalc="{ fn:year-from-date(../d) idivmod 100 }"/>
```

```
</xs:sequence>
```

The entire example in one place:

```
<xs:sequence>
  <xs:sequence dfdl:hiddenGroupRef="tns:hiddenpDate"/>

  <xs:element name="d" type="date">
    <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
      <dfdl:element>
        <dfdl:property name="inputValueCalc">
          {
           fn:date(fn:concat(if(../pdate/yy gt 50) then "19" else "20",
                             if ( ../pdate/yy gt 9 )
                                then xs:string(../pdate/yy)
                                else fn:concat("0",
                                       xs:string(../pdate/yy)),
                                       "-",
                                       xs:string(../pdate/mm),
                                       "-",
                                       xs:string(../pdate/dd)))
          }
        </dfdl:property>
      </dfdl:element>
    </xs:appinfo></xs:annotation>
  </xs:element>
  ...
</xs:sequence>

<xs:group name="hiddenpDate" >
  <xs:sequence>
    <xs:element name="pdate">
      <xs:complexType>
        <xs:sequence dfdl:representation="binary">
          <xs:element name="mm" type="byte"
                 dfdl:outputValueCalc="{ fn:month-from-date(../d) }" />
          <xs:element name="dd" type="byte"
                 dfdl:outputValueCalc="{ fn:day-from-date(../d) }" />
          <xs:element name="yy" type="byte"
                 dfdl:outputValueCalc="{ fn:year-from-date(../d) idivmod 100 }" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>
```

The above sequence contains logically only a single date element.

18 DFDL Expression Language

The DFDL expression language allows the processing of values conforming to the data model defined in the DFDL Infoset. It allows properties in the DFDL schema to be dependent on the value of an occurrence of an element or the value of a DFDL variable. For example, the length of the content of an element can be made dependent on the value of another element in the document.

The main uses of the expression language are as follows:

1. When a DFDL property needs to be set dynamically at parse time from the value of one or more elements of the data. Properties such as dfdl:initiator, dfdl:terminator, dfdl:length, dfdl:occursCount, and dfdl:separator accept an expression.
2. In a dfdl:assert annotation
3. In a dfdl:discriminator annotation to resolve uncertainty when parsing
4. In a dfdl:inputValueCalc property to derive the value of an element in the logical model that doesn't exist in the physical data.
5. In a dfdl:outputValueCalc property to compute the value of an element on unparsing.
6. As the value in a dfdl:setVariable annotation or the dfdl:defaultValue in a dfdl:defineVariable or dfdl:newVariableInstance.

The DFDL expression language is a subset of XPath 2.0 [XPath]. DFDL uses a subset of XML schema and has a simpler information model, so only a subset of XPath 2.0 expressions is meaningful in DFDL Schemas. For example, there are no attributes in DFDL, so the attribute axis is not needed.

XPath 2.0 specification [XPATH2] allows implementation-dependent evaluation of expressions thereby allowing either lazy (sequential) evaluation or full (parallel) evaluation of expressions with OR and AND clauses. This flexibility is not desirable in DFDL 1.0 implementations, so the specification is changed to prescribe lazy (sequential) evaluation left-to-right.

In addition, DFDL expressions never return node-sequences having more than one node. DFDL expressions either return a simple value, a node sequence containing exactly one node/value, or an empty node sequence. Node sequences of length greater than one can be used within the expression, just not as the final result. Alternatively, one can state this as there are no constructs in DFDL which can accept a node sequence of more than one node; hence, DFDL expressions can never return a node sequence of more than one node as their final result.

For nilled elements, an attempt to get the value of a nilled element returns an empty node sequence.

DFDL implementations MUST comply with the error code behaviour in Appendix G of the XPath 2.0 spec and map these to the correct DFDL failure type. All but one of XPath's errors map to a Schema Definition Error. The exception is XPTY0004, which is used both for static and dynamic cases of type mismatch. A static type mismatch maps to a Schema Definition Error, whereas a dynamic type mismatch maps to a Processing Error. A DFDL implementation SHOULD distinguish the two kinds of XPTY0004 error if it is able to do so, but if unable it MUST map all XPTY0004 errors to a Schema Definition Error

Implementation Note: DFDL implementations MAY use off-the-shelf XPath 2.0 processors, but must pre-process DFDL expressions to ensure that the behaviour matches the DFDL specification:

- Ensure that what is returned as the result is not a sequence with length > 1 by appropriate use of fn:exactly-one().
- Check for the disallowed use of those XPath 2.0 functions that are not in the DFDL subset

XPath 2.0 specification [XPATH2] defines its functions to be in namespace http://www.w3.org/2005/xpath-functions. The DFDL specification assumes namespace prefix “fn:” is bound to this namespace.

18.1 Expression Language Data Model

The DFDL expression language operates on the DFDL augmented Infoset with the addition of the hidden elements.

Relative path expressions are evaluated relative to the current Infoset Element Information Item, also referred to as the *current element* for short.

In general, a DFDL expression can only reference an element that precedes the position in the schema where the expression is declared, and it is a schema definition otherwise, with the following exceptions:

- An assert or discriminator on a component may reference an element that is a descendent of the component.
- A dfdl:outputValueCalc property may reference an element that follows the position in the schema where the property is specified.

Implementations MAY have implementation-defined limitations on the use of forward or backward reference or MAY provide controls for bounding the reach of such references. These mechanisms are beyond the scope of this specification.

18.2 Variables

A variable is a binding between a (qualified) name and a (typed) value. Variables are defined using the dfdl:defineVariable annotation (see 7.7); defining a variable causes an initial instance also to be created. Further instances of variables are created using the dfdl:newVariableInstance annotation. Instances of variables are assigned a value using the dfdl:setVariable annotation. Variables are referenced in expressions by preceding the QName with '$'.

This section describes the semantics of variables. Any implementation consistent with the behavior described here is acceptable.

The memory where the information about a variable is stored during DFDL processing is called the *variable memory*. A variable is a name that is associated with a storage tuple in the variable memory.

Specifically, the variable memory contains:

- a counter used to generate locations for new tuples. Initial value is 1.
- an ordered list of locations. Each location contains a tuple of values:
  - has-been-set flag. This Boolean is originally false. dfdl:setVariable changes this flag to true.
  - has-been-referenced flag. This Boolean is originally false. Evaluation of an expression that uses the variable value changes the value to true.
  - has-value flag. This Boolean is originally true if the dfdl:defineVariable or dfdl:newVariableInstance annotation has a default value specified, or if a default value has been supplied externally. Otherwise it is false but is set to true if a dfdl:setVariable annotation is processed.
  - typeID. This string is a type identifier taken from the type specified in the dfdl:defineVariable annotation.
  - value. This is a typed value, or the distinguished value "unknown". The type of the value MUST correspond to the typeID. The value is optionally specified in dfdl:defineVariable or dfdl:newVariableInstance annotations in which case it is referred to as the *default value* for the variable. A default value may also be provided by the DFDL processor when the variable is defined with external "true".

The variable memory is initialized when a dfdl:defineVariable annotation is encountered.

Each time a dfdl:newVariableInstance annotation is encountered, the parser captures the current value of the counter from the variable memory. It then creates a new variable memory where the location counter's value is one greater, and where the list of locations has been augmented with a new tuple at the location given by the prior value of the location counter. The tuple is initialized based on the specifics of the dfdl:defineVariable annotation.

18.2.1 Rewinding of Variable Memory State

Upon exit of the scope where the new variable instance was created, the newly created variable memory is discarded, and the prior variable memory is restored.

Note that the above algorithm ensures that each time a dfdl:newVariableInstance is encountered, a fresh location is initialized for it, and once the scope containing that variable goes out of scope, the instance tuple for the variable can no longer be reached. A different variable instance tuple is then visible.

18.2.2 Variable Memory State Transitions

The flags in the variable memory tuples are interpreted and modified as follows:

DFDL annotation	before annotation processed			after annotation processed		
	has-been-set	has-been-referenced	has-value	has-been-set	has-been-referenced	has-value
defineVariable (without default or external value)	tuple doesn't exist			false	false	false
defineVariable (with default value)	tuple doesn't exist			false	false	true

defineVariable (with external value)	tuple doesn't exist			false	false	true
newVariableInstance (without default value)	tuple doesn't exist			false	false	false
newVariableInstance (with default value)	tuple doesn't exist			false	false	true
setVariable	tuple doesn't exist			Schema Definition Error		
	false	false	false	true	false	true
	false	false	true	true	false	true (also value changed to new value)
	false	true	true	Schema Definition Error – set after reference not allowed.		
	true	any	true	Schema Definition Error – double set not allowed.		
reference variable (from DFDL expression)	tuple doesn't exist			Schema Definition Error		
	false	false	false	Schema Definition Error – undefined variable		
	any	any	true	false	true (value is returned)	true

Table 54 Memory States for Expression Language Variables

The above table describes a set of rules which might be abbreviated as:

- write once, read many
- no write after the value has been read

An exception to this behavior occurs whenever the DFDL processor backtracks because it is processing multiple arms of a choice or as a result of speculative parsing. In this case the variable state is also rewound.

It is a Schema Definition Error if a dfdl:setVariable or a variable reference occurs and there is no corresponding variable name defined by a dfdl:defineVariable annotation.

It is a Schema Definition Error if a dfdl:setVariable provides a value of incorrect type which does not correspond to the type specified by the dfdl:defineVariable.

It is a Schema Definition Error if a variable reference in an expression is able to return a value of incorrect type for the evaluation of that expression. That is, DFDL - including the expressions contained in it - is a statically type-checkable language. DFDL implementations SHOULD issue these Schema Definition Errors prior to processing time if possible.

Even if the errors are detected at processing time, the errors associated with write-after-read, and double-write are Schema Definition Errors because they indicate the schema is not properly designed to use variables consistent with their single-assignment behavior.

18.3 General Syntax

DFDL expressions follow the XPath 2.0 syntax rules but are always enclosed in curly braces "{" and "}".

When a property accepts either a DFDL string literal or a DFDL expression, and the value is a string literal starting with a "{" character, then "{{"must be used to escape the "{" character. Note that no escaping is required on the "}" character.

The syntax "{}" is a Schema Definition Error as it results in an empty XPath 2.0 expression which is not legal. It is not the equivalent of setting the property to empty string.

Examples

```
{ /book/title }
{ $x+2 }
{ if (fn:exists(../field1))  then 1 else 0  }
```

The result of evaluating the expression must be a single atomic value of the type expected by the context, and it is a Schema Definition Error otherwise. Some XPath expressions naturally return a sequence of values, and in this case, it is also Schema Definition Error if an expression returns a sequence containing more than one item.

Additionally:

- Every property that accepts an expression states exactly what the expression is expected to return. To ensure the returned value is of the correct type, an expression must use XPath constructors or the correct literal values.
- What appears lexically as the syntax of an expression follows XPath 2.0 rules. Note specifically that this is not the same as XSD default and XSD fixed property lexical syntax. Specifically, XSD default and XSD fixed properties do not accept expressions. They are always interpreted as XML Schema string literals. See [XSD] for details.
- No extra auto-casting is performed over and above that provided by XPath 2.0. XPath 2.0 has rules for when it promotes types and when it allows types to be substituted. These are in Appendix B.1 of the XPath 2.0 spec.
- If the property is not expecting an expression to return a DFDL string literal, the returned value is never treated as a DFDL string literal.
- If expecting an expression to return a DFDL string literal, the returned value is always treated as a DFDL string literal.
- Within an expression, a string is never interpreted as a DFDL string literal.

18.4 DFDL Expression Syntax

Refer to XML Path Language (XPath) 2.0 [XPath] for a description of XPath expressions

DFDL Expression	::=	"{" Expr "}"
Expr	::=	ExprSingle
ExprSingle	::=	IfExpr | OrExpr
IfExpr	::=	"if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
OrExpr	::=	AndExpr ("or" AndExpr)*
AndExpr	::=	ComparisonExpr ("and" ComparisonExpr)*
ComparisonExpr	::=	AdditiveExpr ((ValueComp) AdditiveExpr)?
AdditiveExpr	::=	MultiplicativeExpr (("+" | "-") MultiplicativeExpr)*
MultiplicativeExpr	::=	IntersectExceptExpr(("*" | "div" | "idiv" | "mod") IntersectExceptExpr)*
IntersectExceptExpr	::=	UnaryExpr (("intersect" | "except") UnaryExpr)*
UnaryExpr	::=	("-" | "+")* ValueExpr
ValueExpr	::=	PathExpr
ValueComp	::=	"eq" | "ne" | "lt" | "le" | "gt" | "ge"
PathExpr	::=	("/" RelativePathExpr?) | RelativePathExpr | FilterExpr
RelativePathExpr	::=	StepExpr (("/") StepExpr)*
StepExpr	::=	AxisStep
AxisStep	::=	(ReverseStep | ForwardStep) Predicate?
ForwardStep	::=	(ForwardAxis NodeTest) | AbbrevForwardStep
ForwardAxis	::=	("child" "::") | ("self" "::")
AbbrevForwardStep	::=	NodeTest | ContextItemExpr
ReverseStep	::=	(ReverseAxis NodeTest) | AbbrevReverseStep

ReverseAxis	::=	("parent" "::")
AbbrevReverseStep	::=	".."
NodeTest	::=	NameTest
NameTest	::=	QName
FilterExpr	::=	PrimaryExpr Predicate?
Predicate	::=	"[" Expr "]"
PrimaryExpr	::=	Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall
Literal	::=	NumericLiteral | StringLiteral
NumericLiteral	::=	IntegerLiteral | DecimalLiteral | DoubleLiteral
VarRef	::=	"$" VarName
VarName	::=	QName
ParenthesizedExpr	::=	"(" Expr ")"
ContextItemExpr	::=	"."
FunctionCall	::=	QName "(" (ExprSingle ("," ExprSingle)*)? ")"

Table 55 DFDL Expression Language

Notes:

1. Only *If* and *path* expression types are supported
2. Only the *child*, *parent*, and *self* axes are supported
3. Predicates are only used to index arrays and so must be integer expressions otherwise a Schema Definition Error occurs
4. A subset of the XPath 2.0 operators is supported
5. NameTest - These QNames are path steps that refer to elements in the DFDL Infoset. If such an element is in a namespace, then the NameTest QName must have a prefix which is bound to the namespace. Specifically, any default namespace is not used to implicitly qualify these NameTest QNames. This behavior is consistent with XPath expression usage in XML Schema [Walmsley] such as in the path property of the xs:selector and xs:field elements within xs:key and xs:unique constraints, and in related XML standards such as XSLT. Note however, that this behavior is different from the way QNames are used in other places in XML and DFDL Schemas such as the ref property of an element reference, or the dfdl:ref property of a DFDL format annotation. There a QName with no prefix must always be referring to a global declaration or definition, and so is augmented with the default namespace when needed.
6. The *FilterExpr* ".[1]" is the same as ".". The *FilterExpr* ".[n]" where n is not equal to 1 returns an empty node sequence.

18.5 Constructors, Functions and Operators

In the function signatures below a '?' following an argument name, argument type or result type indicates that the argument/result can be a node or value of the expected type or it can have no value.

18.5.1 Constructor Functions for XML Schema Built-in Types

The arguments to the constructors are all of type xs:anyAtomicType. Since the expression language can be statically type checked, it is a Schema Definition Error if the type of the argument is not one of the DFDL-supported subtypes of xs:anyAtomicType,

However, many statically type-correct values are still not convertible to the result type. It is a Processing Error if the supplied argument value is not convertible to the constructed type.

The following constructor functions for the built-in types are supported:

Function
xs:string($arg as xs:anyAtomicType) as xs:string
xs:boolean($arg as xs:anyAtomicType) as xs:boolean

xs:decimal($arg as xs:anyAtomicType) as xs:decimal
xs:float($arg as xs:anyAtomicType) as xs:float
xs:double($arg as xs:anyAtomicType) as xs:double
xs:dateTime($arg1 as xs:date, $arg2 as xs:time) as xs:dateTime
xs:time($arg as xs:anyAtomicType) as xs:time
xs:date($arg as xs:anyAtomicType) as xs:date
xs:hexBinary($arg as xs:anyAtomicType) as xs:hexBinary
xs:integer($arg as xs:anyAtomicType) as xs:integer
xs:long($arg as xs:anyAtomicType) as xs:long
xs:int($arg as xs:anyAtomicType) as xs:int
xs:short($arg as xs:anyAtomicType) as xs:short
xs:byte($arg as xs:anyAtomicType) as xs:byte
xs:nonNegativeInteger($arg as xs:anyAtomicType) as xs:nonNegativeInteger
xs:unsignedLong($arg as xs:anyAtomicType) as xs:unsignedLong
xs:unsignedInt($arg as xs:anyAtomicType) as xs:unsignedInt
xs:unsignedShort($arg as xs:anyAtomicType) as xs:unsignedShort
xs:unsignedByte($arg as xs:anyAtomicType) as xs:unsignedByte

Table 56 Basic Constructors

A special constructor function is provided for constructing a `xs:dateTime` value from an `xs:date` value and an `xs:time` value.

Function
fn:dateTime($arg1 as xs:date, $arg2 as xs:time) as xs:dateTime

Table 57 Special Constructor for xs:dateTime

18.5.2 Standard XPath Functions

18.5.2.1 Boolean functions

The following additional constructor functions are defined on the boolean type.

Function	Meaning
fn:true()	Constructs the xs:boolean value 'true'.
fn:false()	Constructs the xs:boolean value 'false'.

Table 58 Boolean functions

The following functions are defined on boolean values. The return type of these functions is xs:boolean.:

Function	Meaning
fn:not($arg?)	If `$arg` is the empty sequence or a node with **[nilled]** true, `fn:not` returns `true`. If `$arg` is a sequence containing a node with **[nilled]** false or **[nilled]** having no value (that is, a node corresponding to a non-nillable element), `fn:not` returns `false`. If `$arg` is a value of type `xs:boolean` or a derived from `xs:boolean`, `fn:not` returns the boolean inverse of `$arg`. If `$arg` is a value of type `xs:string` or a type derived from `xs:string`, `fn:not` returns `true` if the operand value has zero length; otherwise it returns `false`. If `$arg` is a value of any numeric type or a type derived from a numeric type, `fn:not` returns `true` if the operand value is `NaN` or is numerically equal to zero; otherwise it returns `false`. In all other cases, `fn:not` raises a Processing Error. Inverts the xs:boolean value of the argument.

Table 59 Boolean functions

18.5.2.2 Numeric Functions

The following functions are defined on numeric types. Each function returns a value of the same type as the type of its argument. The argument must be convertible to a number type.

Function	Meaning
fn:abs($arg as numeric)	Returns the absolute value of the argument.
fn:ceiling($arg as numeric)	Returns the smallest number with no fractional part that is greater than or equal to the argument.
fn:floor($arg as numeric)	Returns the largest number with no fractional part that is less than or equal to the argument.
fn:round($arg as numeric)	Rounds to the nearest number with no fractional part. When the value is x.5, it rounds toward positive infinity.
fn:round-half-to-even($arg as numeric) fn:round-half-to-even($arg as numeric, $precision as xs:integer)	Takes a number and a precision and returns a number rounded to the given precision. If the fractional part is exactly half, the result is the number whose least significant digit is even.

Table 60 Numeric Functions

18.5.2.3 String Functions

The following functions are defined on values of type `xs:string` and types derived from it. In the functions below which compare strings, DFDL always uses the default Unicode collation algorithm (which is a comparison of codepoint values).

Function	Meaning
fn:concat($arg1 as xs:anyAtomicType, $arg2 as xs:anyAtomicType, ...)	Concatenates two or more xs:anyAtomicType arguments cast to xs:string.
fn:substring($sourceString as xs:string, $startingLoc as xs:double) fn:substring($sourceString as xs:string, $startingLoc as xs:double, $length as xs:double)	Returns the xs:string located at a specified place within an argument xs:string.
fn:string-length($arg as xs:string)	Returns the length of the argument as an xs:integer
fn:upper-case($arg as xs:string)	Returns the upper-cased value of the argument.
fn:lower-case($arg as xs:string)	Returns the lower-cased value of the argument.
fn:contains($arg1 as xs:string, $arg2 as xs:string)	Returns xs:boolean indicating whether one xs:string contains another xs:string.

fn:starts-with($arg1 as xs:string, $arg2 as xs:string)	Returns xs:boolean indicating whether the value of one xs:string begins with the characters of another xs:string.
fn:ends-with($arg1 as xs:string, $arg2 as xs:string)	Returns xs:boolean indicating whether the value of one xs:string ends with the characters of another xs:string.
fn:substring-before($arg1 as xs:string, $arg2 as xs:string)	Returns the characters of one xs:string that precede in that xs:string the characters of another xs:string.
fn:substring-after($arg1 as xs:string, $arg2 as xs:string)	Returns the characters of xs:string that follow in that xs:string the characters of another xs:string.

Table 61 String Functions

18.5.2.4 Date and Time Functions

Function	Meaning
fn:year-from-dateTime($arg as xs:dateTime)	Returns the year from an xs:dateTime value as an xs:integer.
fn:month-from-dateTime($arg as xs:dateTime)	Returns the month from an xs:dateTime value as an xs:integer.
fn:day-from-dateTime($arg as xs:dateTime)	Returns the day from an xs:dateTime value as an xs:integer.
fn:hours-from-dateTime($arg as xs:dateTime)	Returns the hours from an xs:dateTime value as an xs:integer.
fn:minutes-from-dateTime($arg as xs:dateTime)	Returns the minutes from an xs:dateTime value as an xs:integer.
fn:seconds-from-dateTime($arg as xs:dateTime)	Returns the seconds from an xs:dateTime value as an xs:decimal.
fn:year-from-date($arg as xs:date)	Returns the year from an xs:date value as an xs:integer.
fn:month-from-date($arg as xs:date)	Returns the month from an xs:date value as an xs:integer.
fn:day-from-date($arg as xs:date)	Returns the day from an xs:date value as an xs:integer.
fn:hours-from-time($arg as xs:time)	Returns the hours from an xs:time value as an xs:integer.
fn:minutes-from-time($arg as xs:time)	Returns the minutes from an xs:time value as an xs:integer.
fn:seconds-from-time($arg as xs:time)	Returns the seconds from an xs:time value as an xs:decimal.

Table 62 Date and Time Functions

18.5.2.5 Node Sequence Test Functions

The following functions are defined on sequences. (Note that DFDL v1.0 does not support sequences of length > 1 as the final results of expressions.)

In the functions below, if the argument evaluates to the current node, or any enclosing parent node, then it is a Schema Definition Error.

Function	Meaning
fn:empty($arg?)	Indicates whether the provided sequence is empty.
fn:exists($arg?)	Indicates whether the provided sequence is not empty.
fn:exactly-one($arg?)	Returns the input sequence if it contains exactly one item. Raises an error otherwise

fn:count($arg)	Returns the number of items in the value of `$arg` as an xs:integer. Returns 0 if `$arg` is the empty sequence.

Table 63 Node Sequence Test Functions

18.5.2.6 Node functions

This section discusses functions and operators on nodes.

Function	Meaning
fn:local-name() fn:local-name($arg)	Returns the local name of the context node or the specified node as an xs:string.
fn:namespace-uri() fn:namespace-uri($arg)	Returns the namespace URI as an xs:string for the argument node or the context node if the argument is omitted. Returns empty string if the argument/context node is in no namespace.

Table 64 Node functions

18.5.2.7 Nillable Element Functions

This section discusses functions related to nillable elements.

Function	Meaning
fn:nilled($arg?)	Returns an `xs:boolean` true when the argument node Infoset member **[nilled] is** true and false when **[nilled]** is false. If the argument is not an element node, returns the empty sequence. If the argument is the empty sequence, returns the empty sequence. If the argument is an element node and **[nilled]** has no value returns the empty sequence.

Table 65 Nillable Element Functions

18.5.3 DFDL Functions

Function	Meaning
dfdl:contentLength($node, $lengthUnits)	Returns the length of the supplied node's SimpleContent region for elements of simple type, or ComplexContent region for elements of complex type. These regions are defined in Section 9.2 DFDL Data Syntax Grammar. The value is returned as an xs:unsignedLong. The second argument is of type xs:string and must be 'bytes', 'characters', or 'bits' (Schema Definition Error otherwise) and determines the units of length.
dfdl:valueLength($node, $lengthUnits)	Returns the length of the supplied node's SimpleLogicalValue region for elements of simple type, or ComplexValue region for elements of complex type. These regions are defined in Section 9.2 DFDL Data Syntax Grammar. The value is returned as an xs:unsignedLong. For simple types, the dfdl:valueLength() function returns a length which excludes any padding or filling. The second argument is of type xs:string and must be 'bytes', 'characters', or 'bits' (Schema Definition Error otherwise) and determines the units of length.
dfdl:testBit($data, $bitPos)	Returns Boolean true if the bit number given by the xs:nonNegativeInteger $bitPos is set on in the xs:unsignedByte given by $data, otherwise returns Boolean false.
dfdl:setBits($bit1, $bit2, ... $bit8)	Returns an unsigned byte being the value of the bit positions provided by the Boolean arguments, where true is1, false is 0. The number of arguments must be 8.

dfdl:occursIndex()	Returns the position of the current item of an array as an xs:nonNegativeInteger. The first element is at position 1. The function may be used on non-array elements so long as it appears within the dynamic scope of some array element. In this case it returns the index of the current item of the innermost enclosing array element. It is a Schema Definition Error if this function is called when there is no enclosing array element.
dfdl:checkConstraints($node)	Returns boolean true if the specified node value satisfies the XML schema facet constraints that are associated with it. Returns false if the specified node does not meet the constraints or does not exist. The facets that are checked are • minLength, maxLength • pattern • enumeration • maxInclusive, maxExclusive, minExclusive, minInclusive • totalDigits • fractionDigits See Section 5.3 for which facets are checked for each simple type. Additionally, the XSD fixed property is checked. It is a Schema Definition Error if the argument is a complex element.
dfdl:encodeDFDLEntities($arg)	Returns a string containing a DFDL string literal constructed from the $arg string argument. If $arg contains any '%' and/or space characters, then the return value replaces each '%' with '%%' and each space with '%SP;', otherwise $arg is returned unchanged.
dfdl:decodeDFDLEntities ($arg)	Returns a string constructed from the $arg string argument. If $arg contains syntax matching DFDL Character Entities syntax, then the corresponding characters are used in the result. Any characters in $arg not matching the DFDL Character Entities syntax remain unchanged in the result. It is a Schema Definition Error if $arg contains syntax matching DFDL Byte Value Entities syntax.
dfdl:containsDFDLEntities($arg)	Returns a Boolean indicating whether the $arg string argument contains one or more DFDL entities.
dfdl:timeZoneFromDateTime($arg) dfdl:timeZoneFromDate($arg) dfdl:timeZoneFromTime ($arg)	Returns the timezone component, if any, of $arg as an xs:string. The $arg is of type xs:dateTime, xs:date and xs:time respectively. If $arg has a timezone component, then the result is a string in the format of an ISO Time zone designator. Interpreted as an offset from UTC, its value may range from +14:00 to -14:00 hours, both inclusive. The UTC time zone is represented as "+00:00". If the $arg has no timezone component, then "" (empty string) is returned.
dfdl:checkRangeInclusive($node, $val1, $val2) dfdl:checkRangeExclusive($node, $val1, $val2)	Returns boolean true if the specified node value is in the range given by $val1 and $val2. The type of $val1 and $val2 must be compatible with the type of $node, and must be a derivative of xs:decimal, xs:float or

	xs:double. It is a Schema Definition Error if the $node argument is a complex element.

Table 66 DFDL Functions

Notes:

dfdl:valueLength(path, lengthUnits) - returns the value length which excludes any padding or filling which might be added for a *specified length.*

If the element declaration in the DFDL schema corresponding to the Infoset item has the dfdl:inputValueCalc property, then the unpadded length is defined to be 0.

The value length includes the length contributions from introduced escape characters needed to escape contained delimiters (if such are defined and would appear in the output representation).

The value length is also a function of the dfdl:encoding property. Multi-byte and variable-width character set encodings commonly contribute more bytes to the value length than a single-byte character set would.

The value length is computed from the DFDL Infoset value, ignoring the dfdl:length or dfdl:textOutputMinLength property. Other DFDL properties which affect the length of a text or binary representation are respected, it is only an explicit length which is ignored.

For a complex type, this means a bottom up totaling of the dfdl:contentLength() of all the contents and framing of the complex type.

dfdl:contentLength(path, lengthUnits) – returns the length of the content of the Infoset data item as identified by the path argument. This includes padding or filling or truncation which might be carried out for a *specified length* item.

If the element declaration in the DFDL schema corresponding to the Infoset item has a dfdl:inputValueCalc property, then the length is defined to be 0.

When unparsing with dfdl:lengthKind "explicit", the calculation of dfdl:contentLength() returns the value of the dfdl:length property.

For both dfdl:contentLength() and dfdl:valueLength(), the content length excludes any alignment filling as well as excluding any leading or trailing skip bytes. That is, the returned length is about the length of the content, and not about the position of that content in the output data stream.

Use dfdl:encodeDFDLEntities() when the value of a DFDL property is obtained from the data stream using an expression, and the type of the property is DFDL String Literal or List of DFDL String Literals, and the values extracted from the data stream can contain '%' or space characters. If the data already contains DFDL entities, this function should not be used.

The dfdl:decodeDFDLEntities() function is used to create a value which contains characters for which DFDL Character Entities are needed. An example is to create data containing the NUL (character code 0) codepoint. This character code is not allowed in XML documents, including DFDL Schemas; hence, it must be specified using a DFDL Character Entity. Within a DFDL Expression, use this function to obtain a string containing this character.

18.5.4 DFDL Constructor Functions

There is sometimes a need to create a number type from hex binary, and a hex binary type from a number. Accordingly, the following new DFDL specific functions are provided.

Function	Meaning
dfdl:byte ($arg) dfdl:unsignedByte ($arg) dfdl:short ($arg) dfdl:unsignedShort($arg) dfdl:int ($arg) dfdl:unsignedInt ($arg) dfdl:long ($arg) dfdl:unsignedLong ($arg)	These constructor functions behave identically to the XPath 2.0 constructor functions of the same names, with one exception. The argument can be a quoted string beginning with the letter 'x', in which case the remainder of the string is hexadecimal digits that represent a big-endian twos complement representation of a binary number. If the string begins with 'x', it is a Schema Definition Error if a character appears other 0-9, a-f, A-F. Each constructor function has a limit on the number of hex digits, with no more digits than 2, 4, 8, or 16 for the byte, short, int and long versions respectively. It is a Schema Definition Error if more digits are encountered than are suitable for the type being created

dfdl:hexBinary ($arg)	This constructor function behaves identically to the XPath 2.0 constructor function of the same name, with one exception. The argument can also be a long, unsignedLong, or any subtype thereof, and in that case a xs:hexBinary value containing a number of hex digits is produced. The ordering and number of the digits correspond to a binary big-endian twos-complement implementation of the type of the argument. Digits 0-9, A-F are used. The number of digits produced depends on the type of $arg, being 2, 4, 8 or 16. If $arg is a literal number then the type is the smallest signed type (long, int, short, byte) that can contain the value. If a literal number is not able to be represented by a long, it is a Schema Definition Error.

Table 67: DFDL Constructor Functions

Examples:

- dfdl:unsignedInt("xa1b2c3d4") is the unsigned int value 2712847316.
- dfdl:int("xFFFFFFFF") is the signed int value -1.
- dfdl:unsignedByte("xFF") is the unsigned byte value 255.
- dfdl:byte("xff") is the signed byte value -1.
- dfdl:byte("x7F") is the signed byte value 127.
- dfdl:byte("x80") is the signed byte value -128.
- dfdl:unsignedByte("x80") is the unsigned byte value 128.
- dfdl:byte("x0A3") is a Schema Definition Error (too many digits for type).
- dfdl:byte("xG3") is a Schema Definition Error (invalid digit).
- dfdl:hexBinary(xs:unsignedByte(208)) is the hexBinary value "D0".
- dfdl:hexBinary(208) is the hexBinary value "00D0".
- dfdl:hexBinary(-2084) is the hexBinary value "F7DC".

18.5.5 Miscellaneous Functions

Function	Meaning
fn:error() fn:error($id as xs:string) fn:error($id as xs:string, $desc as xs:string, $obj?)	Causes a Processing Error. This function does not return a value. A Processing Error ends the evaluation of the expression. The $id argument is an error code identifier string that distinguishes this error from others. The string should have the structure of an XSD QName; the namespace URI conventionally identifies the component, subsystem, or authority responsible for defining the meaning of the error code, while the local part identifies the specific error condition. This information is incorporated into any diagnostic messages created by the DFDL implementation in response to the Processing Error in an implementation-dependent manner. If the $id argument string does not have the form of an XSD QName, or the QName cannot be interpreted as a meaningful namespace prefix and local identifier, then the Processing Error still occurs but the diagnostic message is created in an implementation-dependent manner. The $desc is a natural-language description of the error condition. This string appears in any diagnostic messages created by the DFDL implementation in response to the Processing Error. The $obj? argument is an arbitrary value used to convey additional information about the error and it is used to construct the diagnostic message in an implementation-dependent manner. If any argument is not supplied the Processing Error occurs but the diagnostic message created is implementation-dependent.

18.6 Unparsing and Circular Expression Deadlock Errors

It is possible for expressions and lengths of elements in a DFDL schema to interact badly, resulting in circular deadlocks. In these cases, an expression is unable to evaluate because it depends in some way on the length of something that depends on the expression itself.

Expression deadlocks are always Schema Definition Errors.

One scenario where such a deadlock can arise is due to what is called the i*nterior-alignment problem*. In this scenario a dfdl:outputValueCalc expression depends on the dfdl:valueLength function being evaluated for a following complex element which due to interior alignments, has a length that depends on its starting position. In this case, a circular deadlock occurs, which is a unparse-time Processing Error.

19 DFDL Regular Expressions

A DFDL regular expression may be specified for the dfdl:lengthPattern format property and the dfdl:testPattern property of the dfdl:assert and dfdl:discriminator annotations. DFDL regular expressions do not interpret DFDL entities.

A DFDL regular expression is defined by a set of valid pattern characters. For portability, a DFDL regular expression pattern is restricted to the inclusive subset of the ICU regular expression [ICURegex] and the Java(R) 7 regular expression [JavaRegex] with the Unicode flags UNICODE_CASE and UNICODE_CHARACTER_CLASS turned on. DFDL regular expressions thereby conform to Unicode Technical Standard #18 , Unicode Regular Expressions, level 1 [UnicodeRegex].

The following regular expression constructs are not common to both ICU and Java(R) 7 and it is a Schema Definition Error if any are used in a DFDL regular expression:

Construct	Meaning	Notes
\N{UNICODE CHARACTER NAME}	Match the named character	ICU only
\X	Match a Grapheme Cluster	ICU only
\Uhhhhhhhh	Match the character with the hex value hhhhhhhh	ICU only
(?# ...)	Free-format comment	ICU only
(?w-w)	UREGEX_UWORD - Controls the behaviour of \b in a pattern	ICU only
(?d-d)	UNIX_LINES - Enables Unix lines mode	Java 7 only
(?u-u)	UNICODE_CASE - Enables Unicode-aware case folding	Java 7 only (1)
(?U-U)	UNICODE_CHARACTER_CLASS - Enables the Unicode version of predefined character classes and POSIX character classes	Java 7 only (2)

Table 68 Disallowed Regular Expression Constructs

Notes:

1. Implementations using Java 7 MUST set flag UNICODE_CASE by default to match ICU.
2. Implementations using Java 7 MUST set flag UNICODE_CHARACTER_CLASS by default to match ICU.

Additionally, the behaviour of the word character construct (\w) is not consistent in ICU and Java 7. In Java 7 \w is

[\p{Alpha}\p{gc=Mn}\p{gc=Me}\p{gc=Mc}\p{Digit}\p{gc=Pc}],

which is a larger set than ICU where \w is

[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}].

The use of \w is not recommended in DFDL regular expressions in conjunction with Unicode encodings, and an implementation MUST issue a warning if such usage is detected.

Character properties are detailed by the Unicode Regular Expressions [UnicodeRegex].

20 External Control of the DFDL Processor

In addition to providing the DFDL schema and data to be parsed or serialized, DFDL Schemas can also be parameterized by external variables.

DFDL processors can provide implementation-defined means to specify:

1. The data to be processed: a data stream when parsing or an Infoset when unparsing.
2. The DFDL schema to be used
3. The *distinguished global element declaration* to be used (specifying both name of element and namespace of that name)
4. Values for external variables

Notice also that a DFDL Schema, like any XML schema, can have multiple top-level element declarations; hence, the distinguished global element declaration is necessary to indicate which of these top-level element declarations is to be the starting point for processing data.

The mechanism by which a DFDL processor is controlled is not specified by this standard. For example, command line DFDL processors MAY use command line options, but DFDL processors embedded in other kinds of software systems may need other mechanisms.

21 Built-in Specifications

For convenience, a standard set of named DFDL format definitions MAY be provided with DFDL processors by implementations. These built-in format definitions may be imported by DFDL schema authors.

22 Conformance

DFDL conformance can be claimed for schema documents and for processors

A schema document conforms to this specification if it conforms to the subset of XML Schema 1.0 defined in Section 5.2 DFDL Subset of XML Schema and consists of components which individually and collectively satisfy all the relevant constraints specified in this document.

Conformance may be claimed separately for a DFDL parser, a DFDL unparser or a DFDL processor that parses and unparses.

1. A DFDL processor claiming conformance MUST identify the level of conformance and version specification claimed.
2. A minimal conforming DFDL processor conforms to this specification when it implements all the non-optional features defined in this document.
3. An extended conforming DFDL processor conforms to the specification when it implements all the non-optional features and some of the optional features defined in this document.
4. A fully conforming DFDL processor conforms to the specification when it implements all the features defined in this document.

See Section 23 Optional DFDL Features for the list of optional features

It is the intention of the DFDL Work Group to provide a conformance test suit to help verify conformance with this specification.

23 Optional DFDL Features

The following table lists the features of the DFDL language that are considered optional for DFDL processor implementations. This list admits very small subsets of the full DFDL specification. For example, a binary-only subset without any expressions or variables is specifically allowed.

Feature	Detection
Validation	External switch
Named Formats	dfdl:defineFormat or dfdl:ref
Choices	xs:choice in the schema
Arrays where size not known in advance	dfdl:occursCountKind 'implicit', 'parsed', 'stopValue'
Expressions	Use of a DFDL expression in any property value
End of parent	dfdl:lengthKind "endOfParent"
Simple type restrictions	xs:simpleType in the schema
Text representation for types other than String	dfdl:representation "text" for Number, Calendar or Boolean types
Delimiters	dfdl:separator <> "" or dfdl:initiator <> "" or dfdl:terminator <> "" or dfdl:lengthKind "delimited"
Nils	XSD nillable 'true' in the schema
Defaults	XSD default or XSD fixed in the schema
Defaulting to Empty String/HexBinary values in the Infoset	dfdl:emptyElementParsePolicy=”treatAsEmpty”
Lengths in Bits	dfdl:alignmentUnits 'bits' or dfdl:lengthUnits 'bits'
Delimited lengths and representation binary element	dfdl:representation 'binary' (or implied binary) and dfdl:lengthKind 'delimited'
Regular expressions	dfdl:lengthKind 'pattern', dfdl:assert with dfdl:testkind 'pattern' , dfdl:discriminator with dfdl:testkind 'pattern'
Zoned numbers	dfdl:textNumberRep 'zoned'
IBM 390 packed numbers	dfdl:binaryNumberRep 'packed'
IBM 390 packed calendars	dfdl:binaryCalendarRep 'packed'
IBM 390 floats	dfdl:binaryFloatRep 'ibm390Hex'
Unordered sequences	dfdl:sequenceKind 'unordered'
Floating elements	dfdl:floating 'yes'
dfdl functions in expression language	DFDL functions in expression
Hidden groups	dfdl:hiddenGroupRef <> ''
Calculated values	dfdl:inputValueCalc <> '' or dfdl:outputValueCalc <> ''
Escape schemes	dfd:defineEscapeScheme in the schema
Extended encodings	Any dfdl:encoding value beyond the core list
UTF-16 Variable Width Characters	dfdl:utf16Width=”variable”
Asserts	dfdl:assert in the schema
Discriminators	dfdl:discriminator in the schema
Prefixed lengths	dfdl:lengthKind 'prefixed'
Variables	dfdl:defineVariable, dfdl:newVariableInstances,

	dfdl:setVariable Variables in DFDL expression language Note that variables as a feature is dependent on the Expressions feature.
BCD calendars	dfdl:binaryCalendarRep "bcd"
BCD numbers	dfdl:binaryNumberRep "bcd"
Multiple schemas	xs:include or xs:import in the schema
IBM 4690 packed numbers	dfdl:binaryNumberRep "ibm4690Packed"
IBM 4690 packed calendars	dfdl:binaryCalendarRep "ibm4690Packed"
DFDL Byte Value Entities	Use of %#r syntax in a DFDL String Literal other than the dfdl:fillByte property
DFDL Standard Character Set Encodings	dfdl:encoding name begins with "X-DFDL-".
Bit Order - Least Significant Bit First	dfdl:bitOrder with value 'leastSignificantBitFirst'

Table 69 Optional DFDL features

In order to provide portability of a DFDL schema, a minimal or extended conforming processor MUST issue warnings about any DFDL properties it does not implement. This warning can simply state that the property is not recognized.

(This allows the implementation to simply have no knowledge of properties it does not need for the subset of features it implements.)

For example, if the hidden groups feature were not implemented, then the implementation most likely would not recognize the dfdl:hiddenGroupRef property at all. Such an implementation MUST issue a warning that the dfdl:hiddenGroupRef property is not recognized.

It is a Schema Definition Error if a DFDL schema uses an optional feature that is not supported by a minimal or extended conforming processor.