(Obsolete) Proposal: Data Layering for Base64, Line-Folding, Compression, Etc

> **Revised per changes on 2021-10-06**
>
> **Describes the feature as-is-built 2018-05-14.**

This memo describes a proposed feature for expressing data layering of pre/post processing operations.

The term "layer" refers to a data stream, but encompasses the notion that the data stream may not be the original source/target of data, but rather can be computed (when parsing) via a transformation on the data. This is called a layering transformation as one data stream is a layer of interpretation on top of a lower level data stream. Layers can generalize to any depth. As with the term "stream", a "layer" does not connote a direction. For parsing one uses an input data layer, for unparsing an output data layer.

For some DFDL purposes, it does not matter whether the underlying data is an original data stream, or a data stream created by way of a layering transformation. In that case we may refer to "the data stream" meaning either the underlying raw data stream, or a data stream created by way of a layered transformation.

Where we must distinguish beween layers, we will use the terms "underlying layer" and "overlying layer" to distinguish the levels.

Most of the discussion here will use parsing as context, but where the unparsing is not clearly symmetric, unparsing will also be described.

New DFDL schema annotations are shown in the "daf:" namespace so as to be clear what are DFDL standard, and what the new proposed extensions are. Our hope would be that these extensions will be suitable for inclusion in a revision of the DFDL standard. (E.g., DFDL v2.0).

The Layering Properties

The following properties are added to dfdl:sequence (with corresponding short forms). They require the "dfdlx" (DFDL extension) namespace prefix.

- layerTransform (literal string) - XSD NCName
- layerEncoding (literal string or DFDL expression)
- layerLengthKind - Can be 'implicit', 'explicit', or 'boundaryMark'.
- layerBoundaryMark (literal string or DFDL expression) - used with dfdl:layerLengthKind 'boundaryMark'
- layerLength (literal string or DFDL expression) - used with dfdl:layerLengthKind 'explicit'

The initial transform names and their supported layerLengthKinds are:

- base64_MIME - layerLengthKind 'boundaryMark' only
- gzip - layerLengthKind 'explicit' only
- lineFolded_IMF - layerLengthKind 'boundaryMark' (without a layerBoundaryMark property - not used. Always CRLF), or layerLengthKind 'implicit' which extends to end of available data.
- lineFolded_iCalendar - same as lineFolded_IMF

An example layer transform is provided also as a test case:

- aisASCIIArmor - layerLengthKind is assumed to be 'boundaryMark' (the property layerLengthKind is ignored), and the boundary mark is assumed to be "," (Comma). This format is the ASCII-armoring used by the AIS (Automated Identification System) format used for ship identification.

Transform names are very specific. They identify not a general class of transformations, but the specifics of the algorithm. For example base64_MIME is the style of base64 encoding/decode used for MIME attachments in Internet messages. It includes limiting the length of lines in the encoded representation to 75 long.

The dfdl:lengthKind 'implicit' means that the transform algorithm itself determines when the encoded data ends. The other length kinds are accompanied by properties allowing isolation of the data that is to be transformed (for parsing).

Note that there is no layerEscapeSchemeRef property - escaping is assumed to not be required by the layering transformation algorithms, or any such behavior would be built-in to the transformation algorithm.

These properties obey all the usual scoping rules. They can appear on dfdl:format annotations so as to be part of named format definitions or even put into lexical scope over a schema file.

The meaning of the values of these properties, and any constraints on those values, are essentially specified by the layer transform algorithm. For example: for layerLengthKind 'boundaryMark' and the base64 layer transform, we expect that the layerBoundaryMark must be a string, and this string has constraints beyond those we see in the the regular dfdl:terminator property. In particular for base64, we expect that layerBoundaryMark **cannot** contain DFDL character entities of any kind. This makes it impossible, for example, for a base64 'boundaryMark' layer to be bounded by a string containing ASCII NUL (character code 0), as there is no way to express this without use of DFDL character entities. These limitations must be enforced by the algorithms themselves.

To show these new annotations at work, a named format that specifies a layering is created via

```
<daf:defineFormat name="base64Format" >
    <dfdl:format dfdlx:layerTransform="base64_MIME" dfdlx:layerLengthKind="boundaryMark" />
</dfdl:defineFormat>
```

These properties are only relevant to xs:sequence constructs, and so a dfdl:ref to a named format using these layer properties is only sensible from a dfdl:sequence or on an xs:sequence.

An xs:sequence where the layerTransform property is defined and non-empty string, is said to be a layered sequence.

Some restrictions apply:

- A layered sequence can have only one child term.
- A layered sequence cannot carry statement annotations (e.g., dfdl:setVariable, dfdl:newVariableInstance, dfdl:assert, etc.)
- All sequence properties other than those beginning with "layer" prefix, are ignored. If they are specified directly on an xs:sequence in short form, or on its dfdl:sequence annotation element, then a warning should be issued.

Layer transforms apply to the SequenceContent grammar region per DFDL Spec Section 9.2.

A layered sequence has a mandatory layer alignment (analogous to mandatory text alignment). This is 1 byte for all currently specified layer transforms; in the future this may change.

A layered sequence has a mandatory length unit. This is 1 byte for all currently specified layer transforms; in the future this may change.

If the length of a layered sequence is needed, for example to store the length of the transformed representation using dfdl:outputValueCalc, then the layered sequence must be enclosed in an element, and the dfdl:contentLength(...) of that element provides the length of the transformed content.

Data Layers as Streams

A data layer is conceptually a stream of bytes. It can be an input layer for parsing, an output layer for unparsing.
Use of the term "stream" here is consistent with java's use of stream as in java.io.InputStream and java.io.OutputStream. These are sources and sinks of bytes. If one wants to decode characters from them you must do so by specifying the encoding explicitly.

A layer transform is a transformation that creates one layer of bytes from another. An underlying layer is encapsulated by a transformation to create an overlying layer.

When parsing, reading from the overlying layer causes reading of data from the underlying layer, which data is then transformed and becomes the bytes of the overlying layer returned from the read.

The layer properties apply to the underlying layer data and indicate how to identify its bounds/length, and if a layer transform is textual, what encoding is used to interpret the underlying bytes.

Some transformations are naturally binary bytes to bytes. Data decompress/compress are the typical example here. When parsing, the overlying layer's bytes are the result of decompression of the underlying layer's bytes.

If a transform requires text, then a dfdl:format encoding must be defined. For example, base64 is a transform that creates bytes from text. Hence, a layer encoding is needed to convert the underlying layer of bytes into text, then the base64 decoding occurs on that text, which produces the bytes of the overlying layer.

We think of some transforms as text-to-text. Line folding/unfolding is one such. Lines of text that are too long are wrapped by inserting a line-ending and either a space or tab. As a DFDL layer transform this line folding transform requires an encoding. The underlying bytes are decoded into characters according to the encoding. Those characters are divided into lines, and the line unfolding (for parsing) is done to create longer lines of data, the resulting data is then encoded from characters back into bytes using the same encoding.

(There may be opportunities to optimize/shortcut these transformations if the overlying layer is the data layer for an element with scannable text representation using the same character set encoding. The re-conversion back to bytes, only to have to then decode bytes to characters of the same encoding again is overhead that can be avoided.)

DFDL can describe a mixture of character set decoding/encoding and binary value parsing/unparsing against the same underlying data representation; hence, the underlying data layer concept is always one of bytes.

(Note: bytes suffices even for mil-std-2045 which can hold a compressed VMF payload. This payload element is always byte aligned even in mil-std-2045, a very bit-oriented format. As of this writing we have no examples of layer transforms that require bit granularity; hence, this is a byte-oriented proposal.)

Daffodil parsing begins with a default standard data input stream. Unparsing begins with a default standard output stream. These are the ultimate underlying layer.

When parsing, left-over data is possible. The decoded layer data may contain extra data that is not consumed by the parse. It is a Parse Error if there is not enough data. Excess data is skipped/ignored.

When unparsing, extra data may have to be created (padding/filling) to satisfy the layer unparsing algorithm. The DFDL schema for the xs:sequence content must create this padded/filled extra data. It is an Unparse Error if the data created when unparsing that is provided to the layer transform encoding algorithm does not satisfy its length requirements.

Parameterization and Computed Results (Checksums, CRC, Parity):

Layer transform algorithms can read and write DFDL Variables. Combining use of a layer with dfdl:newVariableInstance allows one to specify parameters to a particular layering transform, as well as to receive values back from the layer transform. This allows computation of things like checksums, CRCs, or parity across the contents of a layer.

Examples using Data Layering

When a DFDL schema wants to describe say, gzip encoding, then the DFDL annotations might look like this:

```
<annotation><appinfo source="http://www.ogf.org/dfdl/">
  <dfdl:defineFormat name="compressed">
    <dfdl:format dfdlx:layerTransform="gzip" dfdlx:layerLengthKind="explicit" />
  </dfdl:defineFormat>
</appinfo></annnotation>

<sequence dfdl:ref="tns:compressed">
  <group ref="tns:compressedGroupContents" dfdlx:layerLength="{...}" />
</sequence>
```

The above annotation means: when parsing this sequence, take whatever data layer is in effect, layer a gzip data layer on it, and use that until the end of the gzipped data - in this case until the length expressed in the layerLength expression is reached.

If we need to determine or verify the length of the layered data, then we must encapsulate the layered sequence in an element so that a path expression can refer to it.

```
<annotation><appinfo source="http://www.ogf.org/dfdl/">
    <dfdl:defineFormat name="compressed">
      <dfdl:format ref="ex:general" dfdlx:layerTransform="gzip" dfdx:layerLengthKind="explicit" dfdlx:
layerLengthUnits="bytes" />
    </dfdl:defineFormat>
    <dfdl:format ref="ex:general" />
</appinfo></annnotation>

         ...
         <xs:sequence>
          <xs:element name="compressedPayloadLength" type="xs:int" dfdl:representation="binary"
            dfdl:outputValueCalc='{ dfdl:contentLength(../compressedPayload, "bytes") }' />

          <xs:element name="compressedPayload">
            <xs:complexType>
              <xs:sequence dfdl:ref="tns:compressed" dfdlx:layerLength="{ ../compressedPayloadLength }">
                <xs:group ref="tns:compressedGroupContents" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>

          <xs:sequence>
            <xs:annotation>
              <xs:appinfo source="http://www.ogf.org/dfdl/">
                <dfdl:assert>{ compressedPayloadLength eq dfdl:contentLength(compressedPayload, "bytes") }<
/dfdl:assert>
              </xs:appinfo>
            </xs:annotation>
          </xs:sequence>
          <xs:element name="after" type="xs:string" dfdl:lengthKind="delimited" />
        </xs:sequence>
        ...
```

The above illustrates how one obtains length information for layered sequences. The compressed sequence is the complex type model group of the compressedPayload element. The compressedPayloadLength element uses dfdl:outputValueCalc to determine the content length of the compressedPayload element, so that it can be stored when unparsing, and the assertion after the compressedPayload element verifies (when parsing) that the length matches what was stored.

The APIs for defining the gzip, base64, or other transformers enable one to do these transformations in a streaming manner, on demand as data is pulled from the resulting data stream of bytes. Of course it is possible to just convert the entire data object, but we want to enable streaming behavior in case stream-encoded objects are large and an implementation wants to optimize this case.

Let's look at an example of two interacting data layer transforms. Below we have an example that is contrived, but has some resemblance to MIME formats. It parses a textual format where all lines that are too long are folded. WIthin it there are delimiter strings and layer boundaryMark strings (same string actually), computed from the data. The actual data 'payload' here is in the child element named 'body', and it is just a string, but it is base64 encoded. More typically, base64 would be used for binary data, or for text with complex character set encodings. This example is just using a string for exposition purposes.

```
<annotation><appinfo source="http://www.ogf.org/dfdl/">
 <dfdl:defineFormat name="base64">
      <dfdl:format ref="ex:general" dfdlx:layerTransform="base64_MIME" dfdlx:layerLengthKind="boundaryMark"
dfdlx:layerLengthUnits="bytes"
        layerEncoding="iso-8859-1" />
 </dfdl:defineFormat>
 <dfdl:defineFormat name="folded">
      <dfdl:format ref="ex:general" dfdlx:layerTransform="lineFolded_IMF" dfdlx:layerLengthKind="implicit"
dfdlx:layerLengthUnits="bytes"
        layerEncoding="iso-8859-1" />
 </dfdl:defineFormat>
</appinfo></annnotation>

    <xs:element name="root" dfdl:lengthKind="implicit">
      <xs:complexType>
        <xs:sequence dfdl:ref="folded"> <!-- From here, everything is line-folded -->
          <xs:sequence>
            <xs:element name="marker" type="xs:string"
              dfdl:initiator="boundary=" dfdl:terminator="%CR;%LF;" />
            <xs:element name="contents" dfdl:lengthKind="implicit"
              dfdl:initiator="{ fn:concat('--', ../marker, '%CR;%LF;') }">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="comment" type="xs:string"
                    dfdl:initiator="Comment:%SP;" dfdl:terminator="%CR;%LF;" />
                  <xs:element name="contentTransferEncoding"  type="xs:string"
                    dfdl:initiator="Content-Transfer-Encoding:%SP;"
                    dfdl:terminator="%CR;%LF;" />
                  <xs:element name="body" dfdl:lengthKind="implicit" dfdl:initiator="%CR;%LF;">
                    <xs:complexType>
                      <xs:choice dfdl:choiceDispatchKey="{ ../contentTransferEncoding }">
                        <xs:sequence dfdl:choiceBranchKey="base64">
                          <xs:sequence dfdl:ref="tns:base64"
                            dfdl:layerBoundaryMark="{
                              fn:concat(dfdl:decodeDFDLEntities('%CR;%LF;'),'--', ../../marker, '--')
                             }"> <!-- base64_MIME encoding for this sequence -->
                            <xs:element name="value" type="xs:string" />
                          </xs:sequence> <!-- END base64_MIME encoding -->
                        </xs:sequence>
                        <!--
                           This is where other choice branches than base64 would go.
                         -->
                      </xs:choice>
                    </xs:complexType>
                  </xs:element> <!-- END element body -->
                </xs:sequence>
              </xs:complexType>
            </xs:element> <!-- END element contents -->
          </xs:sequence>
        </xs:sequence> <!-- END line folding -->
      </xs:complexType>
    </xs:element>
```

The data corresponding to the above schema is shown here:

```
boundary=frontier%CR;
--frontier%CR;
Comment: This simulates a header field that is so long it will get folded%CR;
 into multiple lines of text because it is too long and my job is at the%CR;
 redundancy department is where I work.%CR;
Content-Transfer-Encoding: base64%CR;
%CR;
TG9yZW0gaXBzdW0gZG9sb3Igc2l0IGFtZXQsIGNvbnNlY3RldHVyIGFkaXBpc2NpbmcgZWxpdCwg%CR;
c2VkIGRvIGVpdXNtb2QgdGVtcG9yIGluY2lkaWR1bnQgdXQgbGFib3JlIGV0IGRvbG9yZSBtYWdu%CR;
YSBhbGlxdWEuIFV0IGVuaW0gYWQ=%CR;
--frontier--
```

The above data uses "%CR;" a DFDL Character Entity, to indicate a literal carriage-return or CR character which is U+0d. When looking at this data, keep in mind that %CR; looks like 4 characters, but is actually only 1.

In the data notice the line initiated by "Comment:". That line has been folded by inserting CRLF before a space, twice to insure no line is longer than 78 characters.

The above data parses to this DFDL infoset - presented as XML. (Apologies for the long lines, but when illustrating line wrapping/folding, they're inevitable.)

```
<ex:root>
  <marker>frontier</marker>
  <contents>
    <comment><![CDATA[This simulates a header field that is so long it will get folded into multiple lines of
text because it is too long and my job is at the redundancy department is where I work.]]></comment>
    <contentTransferEncoding>base64</contentTransferEncoding>
    <body>
      <value><![CDATA[Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad]]></value>
    </body>
  </contents>
</ex:root>
```

In the above the base64 has been decoded into a long string of "Lorem ipsum" nonsense, and the line-folded comment has been unfolded. This data can be unparsed with the same DFDL schema to get back the data representation shown previously. That is to say this data "round trips" through parsing and unparsing.

Example of Multi-layer Transformation

Here's some CSV data

```
last,first,middle,DOB
smith,robert,brandon,1988-03-24
johnson,john,henry,1986-01-23
jones,arya,cat,1986-02-19
```

Here's that data gzipped, which takes 115 byte, pre-pended by a 4-byte integer containing that 115 value (storing the length), then the whole thing base64 encoded:

We'll pre-pend that with a 4-byte binary integer holding the length of 168 which is 4 more bytes: 0000 00A8, then base64 encode all of it:


```
AAAAcx+LCAAAAAAAAAAtyUEKgCAQheG94E1mIDWittG+M0xpaNQIo5tuX0Kb98P7LioVjiTf3sn7
K8CyzlqVO9UIkrcgFTYh9pnBTOOInUPba3XmyOX7WiEGlqfxgJ1B6xpzKEDyEOxUf7JoJq1e/RI4
wXIAAAA=
```


The schema that describes the CSV data without the stream transforms is this:

```
  <xs:element name="file" type="ex:fileType" />

  <xs:complexType name="fileType">
    <xs:sequence>
      <xs:element name="data" dfdl:lengthKind="implicit">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="ex:fileTypeGroup" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

    <xs:group name="fileTypeGroup">
      <xs:sequence dfdl:separator="%NL;" dfdl:separatorPosition="postfix">
        <xs:element name="header" minOccurs="0" maxOccurs="1" dfdl:occursCountKind="implicit">
          <xs:complexType>
            <xs:sequence dfdl:separator=",">
              <xs:element name="title" type="xs:string" maxOccurs="unbounded" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="record" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence dfdl:separator=",">
              <xs:element name="item" type="xs:string" maxOccurs="unbounded" dfdl:occursCount="{ fn:count(../..
/header/title) }"
                 dfdl:occursCountKind="expression" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:group>
```

We can annotate this schema with additional stream transform information to enable it to describe the base64 encoded, compressed data.

One easy way to do this is by modifying the complex type definition for fileType to this:

```
    <xs:complexType name="fileType">
      <!--
           first we have the base64 details
       -->
      <xs:sequence dfdl:ref="ex:base64" dfdlx:layerBoundaryMark="--END--">
        <xs:sequence>
          <!--
              now the gzip details, including the 4-byte gzLength element that stores how long
              the gzipped data is.
           -->
          <xs:element name="gzLength" type="xs:int" dfdl:representation="binary" dfdl:lengthKind="implicit"
            dfdl:outputValueCalc="{ dfdl:contentLength( ../data, 'bytes') }" />
          <!--
             this 'data' element is needed only because we have to measure how big it is when unparsing.
             If we were only worried about parsing, we woundn't need to have this extra 'data' element wrapped
around
             the contents.
           -->
          <xs:element name="data" dfdl:lengthKind="implicit">
            <xs:complexType>
              <!--
                 now the gzipped layered sequence itself
               -->
              <xs:sequence dfdl:ref="ex:gzip" dfdlx:layerLength="{ ../gzLength }">
                <!--
                  finally, inside that, we have the original fileTypeGroup group reference.
                  -->
                <xs:group ref="ex:fileTypeGroup" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
```

Along with that we need the definitions of these named stream formats and default format:

```
    <dfdl:defineFormat name="general">
      <dfdl:format ref="ex:GeneralFormat" lengthKind="delimited" outputNewLine="%CR;%LF;" dfdlx:layerEncoding="
iso-8859-1"
        dfdlx:layerLengthUnits='bytes' />
    </dfdl:defineFormat>

    <dfdl:defineFormat name="base64">
      <dfdl:format ref="ex:general" dfdlx:layerTransform="base64_MIME" dfdlx:layerLengthKind="boundaryMark" />
    </dfdl:defineFormat>

    <dfdl:defineFormat name="gzip">
      <dfdl:format ref="ex:general" dfdlx:layerTransform="gzip" dfdlx:layerLengthKind="explicit" />
    </dfdl:defineFormat>

    <dfdl:format ref="ex:general" />
```

the this schema parses this data, undoing both layers to obtain the expected infoset of:

```
<ex:file>
  <gzLength>115</gzLength>
  <data>
    <header>
      <title>last</title>
      <title>first</title>
      <title>middle</title>
      <title>DOB</title>
    </header>
    <record>
      <item>smith</item>
      <item>robert</item>
      <item>brandon</item>
      <item>1988-03-24</item>
    </record>
    <record>
      <item>johnson</item>
      <item>john</item>
      <item>henry</item>
      <item>1986-01-23</item>
    </record>
    <record>
      <item>jones</item>
      <item>arya</item>
      <item>cat</item>
      <item>1986-02-19</item>
    </record>
  </data>
</ex:file>
```

This schema will round-trip parse then unparse, then parse again, the data.

Summary

- allows stacking transforms one on top of another. So you can have base64 encoded compressed data as the payload representation of a child element within a larger element.
- allows specifying properties of the underlying data layers separately from the properties of the logical data.
- scopes the transforms over a xs:sequence body only.
- Avoids new annotation elements with particulars about scoping.
- Simple: doesn't add new functions for layering use when existing dfdl:contentLength will already handle it.
- Complex cases - e.g., initiator before layered data, are handled by encapsulating the layered sequence in another sequence or element that carries the initiator.
- Layer annotations are only about the determining of the length of the layered region, and the algorithm for transforming the data.
- Layer transforms have mandatory layer alignment (1 byte for now)
- Layer transforms can read DFDL variables for parameters, and write results to DFDL variables.

Open Design Issues

- Debug and trace impact, and how to provide visibility to what is going on when an error occurs in the middle of parsing/unparsing when transforms are in use. E.g., the bit/byte position where a run time parse error occurs would be in some transformed stream, not the underlying stream. I suspect some experience with these transform concepts will be needed before there will be enough information to propose ideas here.

Below is For the Future, once Quoted Printable has been implemented.

VCalendar Example Using Quoted-Printable

Consider this VCALENDAR Data:

```
BEGIN:VCALENDAR
PRODID:
VERSION:1.0
BEGIN:VEVENT
DTSTART:20170903T170000Z
DTEND:20170903T173000Z
LOCATION:test location
UID:040000008200E00074C5B7101A82E0080000000010156B50B224D301000000000000000
    01000000083A43200A4E43F4E800BE12703B99BF0
DESCRIPTION;ENCODING=QUOTED-PRINTABLE:=
 Text that will require line folding: Lorem ipsum dolor sit amet, consecte=
 tur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore=
 magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco=
 laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor i=
 n reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla par=
 iatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui =
 officia deserunt mollit anim id est laborum.=0D=0A=0D=0A =0D=0A=0D=0A=0D==
 =0A
SUMMARY:test subject
PRIORITY:3
END:VEVENT
END:VCALENDAR
```

We want to create a schema that describes this.

In the above there are two behaviors that require use of stream transforms. First is the UID. This has been broken to a maximum line length of 76 characters by way of the folded-lines transformation.

The second is the DESCRIPTION which uses a transformation called QUOTED-PRINTABLE which both achieves short line lengths, and also enables embedding of CR, LF, and other characters at the ends of lines.

The result is that we want this XML Infoset:

```
<VCalendar>
  <ProdID>-//Microsoft Corporation//Outlook 15.0 MIMEDIR//EN</ProdID>
  <Version>1.0</Version>
  <VEvent>
    <DTStart></DTStart>
    <DTEnd></DTEnd>
    <Location>test location</Location>


<UID>040000008200E00074C5B7101A82E0080000000010156B50B224D3010000000000000000001000000083A43200A4E43F4E800BE12703B
99BF0</UID>
    <Description>
      <Encoding>QUOTED-PRINTABLE</ENCODING>
      <QP/>

 <Value>Text that will require line folding: Lorem ipsum dolor sit
amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.&#xEOOD;
&#xEOOD;
 &#xEOOD;
&#xEOOD;
&#xEOOD;
</Value>
    </Description>
    <Summary>test subject</Summary>
    <Priority>3</Priority>
  </VEvent>
</VCalendar>
```

Notice the CRLFs at the end. The CRs are represented as remapped to Private-Use-Area(PUA) E00D entities.

The DFDL schema for this, including the specification of the layering transform behaviors is below. This assumes a hypothetical layerLengthKind of 'pattern'.

```
<xs:schema ....>

 <dfdl:format separatorPosition="infix" dfdlx:layerLengthKind="boundaryMark" encoding="utf-8"
  occursCountKind="parsed" separator="" sequenceKind="ordered"
  separatorPosition="infix"/>

 <dfdl:defineFormat name="folded">
  <dfdl:format dfdlx:layerTransform="foldedLines" dfdlx:layerLengthKind="boundaryMark" dfdlx:layerEncoding="us-
ascii"/>
  <!-- boundaryMark here means to enclosing end-of-data, as no boundary mark delimiter is defined. -->
</dfdl:defineFormat>

<dfdl:defineFormat name="qp">
  <dfdl:format dfdlx:layerTransform="quotedPrintable" dfdlx:layerLengthKind="pattern"
      dfdlx:layerLengthPattern="[^\n]*?(?=(?<!=)\n)"/>

 <!-- QPs are terminated by a newline that is not preceded by an =.
      This final newline is not consumed as part of the content. -->

 <!-- Alternatively, the QP transform itself can determine the length
      by searching for this final newline (but leaving it there).
      In which case the lengthKind would be "implicit" -->
</dfdl:defineFormat>

 <xs:element name="VCalendar" dfdl:initiator="BEGIN:VCALENDAR%NL;" dfdl:terminator="END:VCALENDAR%NL; END:
VCALENDAR">
  <xs:complexType>
    <xs:sequence dfdl:separator="%NL;" dfdl:sequenceKind="unordered">
      <xs:sequence dfdl:ref="tns:folded">
         <xs:element name="ProdID" type="xs:string" dfdl:initiator="PRODID:" minOccurs="0"/>
      </xs:sequence>
      <xs:element name="Version" type="xs:string" dfdl:initiator="VERSION:" minOccurs="0" />
      <xs:element name="VEvent" maxOccurs="unbounded" minOccurs="0" dfdl:occursCountKind="parsed"
        dfdl:initiator="BEGIN:VEVENT%NL;" dfdl:terminator="END:VEVENT">
        <xs:complexType>
          <xs:sequence dfdl:separator="%NL;" dfdl:sequenceKind="unordered">
            <xs:element name="DTStart" type="xs:string" dfdl:initiator="DTSTART:" />
            <xs:element name="DTEnd" type="xs:string" dfdl:initiator="DTEND:" />
            <!--
              content from here could have long lines, so must be folded
            -->
            <xs:sequence dfdl:ref="tns:folded">
              <xs:element name="Location" type="xs:string" dfdl:initiator="LOCATION:" minOccurs="0"/>
              <xs:element name="UID" type="xs:string" dfdl:initiator="UID:" minOccurs="0"/>
              <xs:element name="Description" dfdl:initiator="DESCRIPTION:" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                   <xs:element name="Encoding" type="xs:string"
                               dfdl:initiator="ENCODING=" dfdl:terminator=":" minOccurs="0" />
                     <xs:choice dfdl:choiceDispatchKey="{ if (fn:exists(./Encoding)) then ./Encoding else '' }">
                       <!--
                         we inspect the value of the Encoding element and decide what branch of the choice
                         based on it
                        -->
                       <xs:sequence dfdl:choiceBranchKey="QUOTED-PRINTABLE">
                         dfdl:separator="" dfdl:sequenceKind="unordered">
                         <!--
                          Each branch starts with a distinct dummy element to satisfy the UPA rules of XML
Schema
                         -->
                         <xs:element name="QP" type="xs:string" dfdl:inputValueCalc="{ '' }" />
                         <!--
                          Here notice that the layerRef for the qp data is scoped to just this inner element.
                         -->
                         <xs:sequence dfdl:ref="tns:qp">
```

```
                        <xs:element name="Value" type="xs:string"/>
                      </xs:sequence><!-- end layer quoted printable -->
                    </xs:sequence>
                    <!--
                       repeat the above pattern for the choice branches for the various encodings
                     -->
                  </xs:choice>
                 </xs:sequence>
               </xs:complexType>
             </xs:element>
             <xs:element name="Summary" type="xs:string"  dfdl:initiator="SUMMARY:" minOccurs="0"/>
             <xs:element name="Priority" type="xs:string" dfdl:initiator="PRIORITY:" minOccurs="0" />
           </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:sequence><!-- end folded layer -->
   </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```