Layers - Algorithmic Extensions for DFDL

Contents

1 Introduction

This page describes a DFDL language extension known as *Layers*.

A *layer* is an algorithmic transformation of the data stream that cannot be expressed using regular DFDL properties. When parsing it is like a pre-processing of the data stream which happens before parsing. When unparsing it is like a post-processing of the data stream which happens after unparsing. The layer can underlie a part of the data stream, or all of it.

Layers are *byte oriented*. The smallest non-zero layer size is a single byte, and the length of all layers are in bytes. Throughout this discussion of layers, when length is discussed, the units of measure are always bytes.

Layers can be composed together, that is, with one on top of another, and their composition is orthogonal (as in non-interacting). This is actually very common. For example, binary data being included into a textual data format can be both gzip compressed, and then base64 encoded, which requires 2 layers to describe. There is no limit to this depth.

In the section on Using Layers below we will look at an example that uses multiple layers together.

1.1 Built-in Layers

Daffodil includes several built-in layers: - base64_MIME - fourbyteswap - twobyteswap - gzip - lineFolded_IMF - lineFolded_iCalendar

Daffodil also includes two utility layers that are used in combination with other layers to isolate the subset of the data stream the layer algorithm will operate upon. These are: - boundaryMark - fixedLength

Each of the built-in layers will be documented separately below with examples of their usage.

1.2 Custom Plug-In Layers

Additional layers can be written in Java or Scala and deployed as *plug-ins* for Daffodil. These are generally packaged as DFDL *layer schemas*, a kind of *component schema*, that provide the layer packaged for import by other DFDL *assembly* schemas that use the layer in the data format they describe.

1.3 Layer Kinds: Transforming Layers and Checksum Layers

There are two different kinds of layers, though they share many characteristics. They are *transforming* layers, and *checksum* layers. Both run small algorithms over part (or all) of the data stream. The difference is the purpose of the algorithm and its output.

1.3.1 Transforming Layers

These layers decode data (when parsing), and encode data (when unparsing). The simplest example of a transforming layer is the `base64_MIME` layer which decodes the well known base64 encoding which is commonly used to encode binary data inside textual data formats.

Besides decoding and encoding the data stream for the parser/unparser, transforming layers can be parameterized using DFDL variables. They can also assign computed result values to DFDL variables, though this is uncommon.

Custom transforming layers are created by deriving an implementation from the Daffodil API's `Layer` class which is introduced in a later section.

1.3.2 Checksum Layers

Checksum layers are a simplified kind of layer which do not decode or encode data, they simply pass through the data unmodified, but while doing so they compute a checksum, hash, or Cyclic Redundancy Check (CRC) over the data stream.

The value of the checksum (or hash or CRC) is assigned to a DFDL variable as the result of the layer. This makes the value available for use by the DFDL schema that uses the checksum layer. When parsing, the value of this DFDL variable can then be compared to a checksum field in the data, and either an invalid data element or an parse-error can be created if the checksum in the data stream does not match the computed value. When unparsing, the value of this DFDL variable can be written to an element using the `dfdl:outputValueCalc` property.

An example of a checksum layer plug-in is in the EthernetIP DFDL schema, which uses a Daffodil layer to describe the IPv4 packet header checksum algorithm.

Custom checksum layers are created by deriving an implementation class from the Daffodil API's `ChecksumLayer` class, which is introduced in a later section.

2 Using Layers

To use a layer you must know - the layer's namespace URI - the layer's name - the names of any layer parameter variables - the names of any layer result variables

Each layer normally has a *layer schema file* (.dfdl.xsd) which contains the DFDL variable's `dfdl:defineVariable` declarations (if any).

A `sequence` group encloses the scope, within the DFDL schema, of the layer. Often layers are used to describe checksums or decoding/encoding on small regions within a larger data format.

The DFDL extension property `dfdlx:layer` declares that the data underlying a `sequence` is to use a layer. The value of the property is a QName identifying the namespace (via prefix) and name of the layer.

A layer's namespace may define DFDL variables that are used to pass parameters to the layer, or to receive results back from the layer (usually for checksum layers).

Lastly, there is the concept of *layer length limiting*.

2.1 Layer Length Limiting

A layer can process an entire input file/stream. Such a layer is said to be *unlimited*.

A layer can also have a specific data length built into it. The `IPv4Checksum` layer (in the aforementioned EthernetIP DFDL schema) has a fixed length of 20 bytes. Such a layer is said to be *self limiting*.

More commonly, a layer has a limited region within the data stream that it is supposed to describe. The restriction of the layer so that it only describes the expected part of the data is called *layer limiting*.

There are these ways to do layer limiting, that differ in important ways: - using enclosing elements of specified length - using a Utility Layer that is built-in to Daffodil - binding a DFDL variable specific to a particular layer

2.1.1 Layer Limiting using Enclosing Elements of Specified Length

If the `sequence` for a layer is the model group of the complex type of an element, and that element has specified length (meaning it has `dfdl:lengthKind` property of `explicit`, `prefixed` or `pattern`), then the length of the element limits the length of the layer within it.

In this case the length of the layer is limited when parsing, but it is NOT limited when unparsing.

2.1.2 Layer Limiting using the Utility Layers

Some layers do not decode or encode data but only restrict the length for other layers, for example by using a length layer variable or delimiter. See the `boundaryMark` layer and `fixedLength` layer for examples.

2.1.3 Layer Limiting by Binding a DFDL Variable

Some layers will define a DFDL variable that must be set to a non-negative integer value to specify the layer length that the layer will use. All checksum layers built with the Daffodil API `ChecksumLayer` base class, use the DFDL `layerLength` variable to specify the length (in bytes) of the layer.

Layers may specify restrictions on the minimum and maximum allowed values of these lengths, and passing an out-of-range value for the variable is a processing error.

3 Example: Line Folding

Consider the line folding layer, specifically the `lineFolded_IMF` layer, which is built-in to Daffodil.

Line folding is a way of encoding textual data formats so that no line of text is longer than a limited line length.

Consider this data :

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
 tempor incididunt ut labore et dolore magna aliqua. Ut enim ad Lorem
 ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
 tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
```

This data has been *line folded* at roughly 72 characters by inserting a CRLF before an existing space in the data. Each line ends with a CRLF (`\r\n`) and the second through fourth lines begin with a space as a way of indicating that they are extension lines. This data is supposed to be reassembled to form a long single-line string by removing all CRLF pairs.

The result should be this single longer string which does not contain any line endings:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
    tempor incididunt ut labore et dolore magna aliqua. Ut enim ad Lorem
    ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
    tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
```

To achieve this we would use the `lineFolded_IMF` layer. This layer has a specific namespace which our DFDL schema will define a prefix `lf` for like this:

```
<schema xmlns:lf="urn:org.apache.daffodil.layers.lineFolded" ...>
```

Our DFDL schema will import the layer schema for the line-folded layer with this import:

```
<import namespace="urn:org.apache.daffodil.layers.lineFolded"
        schemaLocation="/org/apache/daffodil/layers/xsd/lineFoldedLayer.
           dfdl.xsd"/>
```

Then the layer is incorporated into our DFDL schema like this:

```
<sequence dfdlx:layer="lf:lineFolded_IMF">
  ... elements to be parsed from the unfolded layer data go here ...
</sequence>
```

You can see that use of a layer is described using the `dfdlx:layer` property, and the specific layer is identified by a QName using the previously defined namespace prefix. The scope of the layer is the duration of the sequence it appears on. The `dfdlx:layer` property can *only* be used on an XSD `sequence`.

The `lineFolded_IMF` layer does not define any DFDL variables in its namespace as it has no parameters and produces no results. As of Daffodil 3.8.0, the `import` statement above is optional for the line-folded layers as the DFDL schema file `lineFoldedLayer.dfdl.xsd` does not contain any definitions. In the future however, parameters may be added, so for uniformity all layers define a DFDL schema to be imported as part of using the layer.

If the line-folding applies to the entire data stream (not uncommon for formats that use line folding), then what has been described is all one needs to describe the layer aspects of a data format. Other examples will show how the layer length can be limited to a sub-region of the data.

More detailed documentation for the Line Folded Layers is below.

4 Example: Base64, GZip, and BoundaryMark Layers used Together

In this example, the data consists of a preliminary string, a section of CSV-like data, and a final string element. The CSV-like data section in the middle is both gzipped, and then base64 encoded, with the boundary value mark of "`=_END_=`" indicating where the base64 part ends.

The point of the preliminary string and the final string is to show that this sort of layering composes properly with surrounding data elements in the schema. Those strings could be anything at all, they're just strings to make this example simpler.

The physical data string/stream looks like this:

```
example of data before the base64 gzipped part
H4sIAAAAAAAA/y3JQQqAIBCF4b1nmYG0iNxG+84wpqFhI4xuun0Gbd4P78tUG5xJ+t7J+
    xxg21dV
79QiSHFBGjgh9oVB22XBYUQzqatErv36CjGwPJ/0OGg0Y1c0FUgegoPaLwa1VS9htd+UbgAAAA
    ==
=_END_=
example of data after the base64 gzipped part
```

That base64 encoded part, if decoded and decompressed looks like this CSV-like data:

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

The Infoset, presented as XML, that we will create for this entire data stream using Daffodil parsing looks like:

```
<ex:data>
  <before>example of data before the base64 gzipped part</before>
  <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>
  <after>example of data after the base64 gzipped part</after>
</ex:data>
```

With that characterization of the problem, let's look at the DFDL schema with the Daffodil layering properties to do the boundaryMark, base64 decode, and gzip decompression. (Note that each of these layers is described further in its own section below on this page.)

First there is the start of the schema which introduces namespace prefix definitions for DFDL, for the DFDL extensions (dfdlx) prefix which is used for the `dfdlx:layer` property.

This also sets up 3 namespace prefixes for the namespaces of the boundaryMark (bm), gzip (gz) and base64_MIME (b64) layers which will be imported just after this header.

```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:dfdlx="http://www.ogf.org/dfdl/dfdl-1.0/extensions"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:bm="urn:org.apache.daffodil.layers.boundaryMark"
  xmlns:b64="urn:org.apache.daffodil.layers.base64_MIME"
  xmlns:gz="urn:org.apache.daffodil.layers.gzip"
  xmlns:ex="http://example.com"
  targetNamespace="http://example.com">
```

Next we import the Daffodil built in general format which provides most DFDL properties with sensible starting values, and import the namespaces of our boundaryMark, gzip, and base64_MIME layers, and lastly, set the default format properties for use by this schema file:

```
<include schemaLocation="/org/apache/daffodil/xsd/DFDLGeneralFormat.dfdl.
   xsd"/>

<import namespace="urn:org.apache.daffodil.layers.boundaryMark"
        schemaLocation="/org/apache/daffodil/layers/xsd/boundaryMarkLayer.
           dfdl.xsd"/>
<import namespace="urn:org.apache.daffodil.layers.gzip"
        schemaLocation="/org/apache/daffodil/layers/xsd/gzipLayer.dfdl.xsd
           "/>
<import namespace="urn:org.apache.daffodil.layers.base64_MIME"
        schemaLocation="/org/apache/daffodil/layers/xsd/base64_MIMELayer.
           dfdl.xsd"/>
```

```
<annotation><appinfo source="http://www.ogf.org/dfdl/">
  <dfdl:format ref="ex:GeneralFormat" representation="binary"/>
</appinfo></annotation>
```

The root element of the schema is `data`, and this element has type `dataType`. (A separate named complex type definition for the root element is a recommended style for DFDL schemas as it allows someone else to reuse the schema without getting involved in element namespace prefix complexities.) The `dataType` just shows us the `before` and `after` elements, and references a group named `layeredDataGroup` for the layered part in between.

```
<element name="data" type="ex:dataType"/>

<complexType name="dataType">
  <sequence dfdl:separator="%NL;">
    <element name="before" type="string" dfdl:lengthKind="delimited"/>
    <group ref="ex:layeredDataGroup"/>
    <element name="after" type="string" dfdl:lengthKind="delimited"/>
  </sequence>
</complexType>
```

Now we see the definition of the layering. All 3 layers are declared in this named group. The lines here are numbered so that the text that follows can refer to them:

```
1 <group name="layeredDataGroup">
2   <sequence dfdlx:layer="bm:boundaryMark">
3     <annotation>
4       <appinfo source="http://www.ogf.org/dfdl/">
5         <dfdl:newVariableInstance ref="bm:layerEncoding" defaultValue="
   iso-8859-1"/>
6         <dfdl:newVariableInstance ref="bm:boundaryMark" defaultValue="=
   _END_="/>
7       </appinfo>
8     </annotation>
9     <sequence dfdlx:layer="b64:base64_MIME">
10       <sequence dfdlx:layer="gz:gzip">
11         <group ref="ex:dataFieldsGroup"/>
12       </sequence>
13     </sequence>
14   </sequence>
15 </group>
```

In the above, at line 2 we see the sequence declares the boundaryMark layer. This layer requires 2 parameters to be passed by binding DFDL variables, which occurs on lines 5 and 6. Line 5 binds the `layerEncoding` variable and note the prefix since DFDL variable names can be in a specific namespace. Line 6 binds the `boundaryMark` variable to the string "`=_END_=`". Note that the variable and the layer both are named `boundaryMark`, but this is just coincidence. Layers do not have to have parameters that match their name, nor even any parameters at all. The boundary mark is the string which marks the end of the data based on scanning for that ending mark.

The data region now being isolated, the `boundaryMark` layer contains just one sequence, and line 9 which itself declares that the data is base64 encoded by specifying the base64_MIME layer. This layer has no parameters, so there are no `dfdl:newVariableInstance` statements to bind any variables. The region of data that is base64 encoded is then described by the sequence at line 10 which declares that the data is compressed by the `gzip` algorithm.

Finally, at line 11 we reference a group which describes what the data format of the underlying CSV-like data. That group's definition is regular DFDL without any of the complexity of layers. This is useful for testing, as data can be parsed using this group (in the complexType of an element) to ensure it works properly without involving the complexity of layers. This group definition is the last thing in the schema:

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

</schema>
```

The above schema works both to parse, but also to unparse this data.

5 Using Custom Plug-In Layers

A custom plug-in layer is used in the same manner as the built-in Daffodil layers with just a few additional details: 1. The custom layer will have a DFDL schema that declares any DFDL variables it uses, and defines any data formats needed to facilitate use of the layer. This schema will need to be included/imported. Normally this schema would be a small single-file schema and it can be loaded from the classpath. 2. The custom layer's compiled Java/Scala code is dynamically loaded using the Java Service Provider Interface (SPI). Hence, they must be compiled into Jar files with specific META-INF metadata enclosed, and these jars must appear on the Java CLASSPATH so that they can be found and loaded.

The layer API is defined via Java clases and interfaces to enable writing of custom layers in either Java or Scala.
(One of the built-in layers (Gzip) is written in Java now, by way of proving that one can write a Layer in Java.)

Transformer layer classes are derived from the `Layer` base class.

Checksum layer classes are derived from the `ChecksumLayer` base class.

Further details on how to define custom plug-in layers is in the Javadoc for the Layer API

5.1 —-

6 Daffodil Built-In Layer Documentation

Each of the layers built-in to the Daffodil implementation are documented in a section below which gives the name, namespace, variables, and some usage notes.

The built-in layers are: - base64_MIME - fourbyteswap - twobyteswap - gzip - lineFolded_IMF - lineFolded_iCalendar

---

6.1 Base64 MIME Layer

- Name: base64_MIME
- Namespace URI: urn:org.apache.daffodil.layers.base64_MIME
- Parameter Variables: None
- Result Variables: None
- Import Statement:

```
<xs:import namespace="urn:org.apache.daffodil.layers.base64_MIME"
    schemaLocation="/org/apache/daffodil/layers/xsd/base64_MIMELayer.
        dfdl.xsd"/>
```

This uses the standard `java.util.Base64` classes, specifically the MIME encoding/decoding.

This is specified by RFC 2045. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return `\r` followed immediately by a linefeed `\n` as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

6.2 BoundaryMark Layer

- Name: boundaryMark
- Namespace URI: urn:org.apache.daffodil.layers.boundaryMark
- Parameter Variables
  - `boundaryMark` a string which delimits the end of the layer data.
    - No escape schemes are applied.
    - DFDL character entities and character class entities are not allowed.
  - `layerEncoding` a string which names a character set encoding which is used in the scanning for the boundary mark.
- Result Variables: None
- Import Statement:

```
<xs:import namespace="urn:org.apache.daffodil.layers.boundaryMark"
   schemaLocation="/org/apache/daffodil/layers/xsd/boundaryMarkLayer
      .dfdl.xsd"/>
```

Isolates text data by way of a boundary mark string. This string is a delimiter of the layer data when parsing. The data up to but not including the boundary mark string becomes the data available for parsing. On completion of the layer parse, the parse position is after the boundary mark string. When unparsing the string is inserted after the (otherwise unbounded length) layer data.

This is like a terminating delimiter of a DFDL element or sequence/ choice group, except that there is no escaping mechanism, so the data cannot in any way contain the boundary mark string. Furthermore, elements within the layer can have any `dfdl:lengthKind`, but this does not affect the search for the boundary mark.

For example, when using an element with `dfdl:lengthKind="delimited"` and a `dfdl:terminator="END"`, if that element contains a child element with `dfdl:lengthKind="explicit"`, then the search for the “END” terminator is suspended for the length of the child element, and that search resumes after the child element’s length has been parsed.

In contrast to this, if a boundary mark layer is used with the `boundaryMark` variable bound to “END”, then the data stream is decoded as characters in the charset encoding given by the `layerEncoding` variable, and the layer continues until the “END” is found. The `dfdl:lengthKind` of any child element enclosed within the layer, or even the lengths of other layers found within the scope of this boundary mark layer are not considered and do not disrupt the search for the boundary mark string.

6.3 Byte-Swapping Layers

- Layer Names:
  - `twobyteswap`
  - `fourbyteswap`
- Namespace URI: urn:org.apache.daffodil.layers.byteSwap
- Parameter Variables:
  - `requireLengthInWholeWords` - an `xs:string` which can be "yes" or "no". Defaults to "no". Indicates whether it is a processing error if the layer length turns out to not be a multiple of the word size. If bound to a string other than "yes" or "no" it is a Schema Definition Error.
- Result Variables: None
- Import Statement:

```
<xs:import namespace="urn:org.apache.daffodil.layers.byteSwap"
   schemaLocation="/org/apache/daffodil/layers/xsd/byteSwapLayer.dfdl
       .xsd"/>
```

Layers that re-order bytes according to the word size which is 2 for `twobyteswap` and 4 for `fourbyteswap` respectively. These layers implement streaming behavior, meaning they do not require buffering up the data; hence, they can be used on very large data objects. Bytes within the wrapped input stream are re-ordered *word size* bytes at a time.

For example, with the `requireLengthInWholeWords` as "no" (the default), if the wrapped input stream contains 10 bytes and word size is 4, then the bytes from the wrapped input stream are returned in the order 4 3 2 1 8 7 6 5 10 9. Note that the last 4-byte word is incomplete, but the 2 available bytes are re-ordered anyway. If wordsize were 2 then the bytes from the wrapped input stream are returned in the order 2 1 4 3 6 5 8 7 10 9.

If `requireLengthInWholeWords` is bound to "yes", then if the length is not a multiple of the word size a processing error occurs.

6.4 FixedLength Layer

- Name: fixedLength
- Namespace URI: urn:org.apache.daffodil.layers.fixedLength
- Parameter Variables
  - `fixedLength` an `unsignedInt` which gives the length, in bytes, of the layer data. This must be in the range from 0 to 32767 inclusive. It is a processing error otherwise. This length is enforced on both parsing and unparsing the layer.

- Result Variables: None
- Import Statement:

```
<xs:import namespace="urn:org.apache.daffodil.layers.fixedLength"
   schemaLocation="/org/apache/daffodil/layers/xsd/fixedLengthLayer.
      dfdl.xsd"/>
```

Suitable only for small sections of data, not large data streams or large files. The entire fixed length region of the data will be pulled into a byte buffer in memory.

6.5 GZIP Layer

- Name: gzip
- Namespace URI: urn:org.apache.daffodil.layers.gzip
- Parameter Variables: None
- Result Variables: None
- Import Statement:

```
<xs:import namespace="urn:org.apache.daffodil.layers.gzip"
   schemaLocation="/org/apache/daffodil/layers/xsd/gzipLayer.dfdl.
      xsd"/>
```

This layer uses the `java.util.zip.GZIPInputStream` and `java.util.zip.GZIPOutputStream` libraries to decode and encode.

Prior to Java 16, the `java.util.zip.GZIPOutputStream` wrote a value of zero for the OS field in the header (byte index 9). In Java 16, this was changed to a value of 255 to better abide by the GZIP specification. Unfortunately, this means unparsed data using a GZIP layer might have a single byte difference, depending on the Java version used. To avoid inconsistent behavior of test failures that expect a certain byte value this layer always writes a consistent header (header byte 9 of 255) regardless of the Java version.

6.6 Line Folded Layers

- Layer Names:
  - lineFolded_IMF - conforms to IETF RFC 2822 Internet Message Format (IMF)
  - lineFolded_iCalendar - conforms to IETF RFC 5545 iCalendar
- Namespace URI: urn:org.apache.daffodil.layers.lineFolded
- Parameter Variables: None
- Result Variables: None
- Import Statement:

```
        <xs:import namespace="urn:org.apache.daffodil.layers.lineFolded"
            schemaLocation="/org/apache/daffodil/layers/xsd/lineFoldedLayer.
                dfdl.xsd"/>
```

6.6.1 General Usage

There is a limitation on the compatibility of line folding of data with adjacent parts of the format which also use line-endings. For example, line folding can interact badly with surrounding elements of `dfdl:lengthKind 'pattern'` if the pattern is, for example `".*?\\r\\n(?!(?:\\t|\\ ))"` which is anything up to and including a CRLF not followed by a space or tab. The problem is that line folding converts isolated `\n` or `\r` into `\r\n`, and if this just happens to be followed by a non space/tab character this will have inserted an end-of-data in the middle of the data.

6.6.2 Layer Name: lineFolded_IMF

For IMF, unfolding simply removes CRLFs if they are followed by a space or tab.

When unparsing, the folding is more complex, as CRLFs can only be inserted before a space/tab that appears in the data. If the data has no spaces, then no folding is possible. If there are spaces/tabs, the one closest to (and before) position 78 is used unless it is followed by punctuation, in which case a prior space/tab (if it exists) is used. (This preference for spaces not followed by punctuation is optional, it is not required, but is preferred in the IMF RFC.)

Note: folding is done by some systems in a manner that does not respect character boundaries - i.e., in utf-8, a multi-byte character sequence may be broken in the middle by insertion of a CRLF. Hence, unfolding initially treats the text as iso-8859-1, i.e., just bytes, and removes CRLFs, then subsequently re-interprets the bytes as the expected charset such as utf-8.

IMF is supposed to be US-ASCII, but implementations have gone to 8-bit characters being preserved, so the above problem really can occur.

IMF has a maximum line length of 998 characters per line excluding the CRLF.

- *WARNING This check for 998 characters (or longer) is not implemented (As of Daffodil version 3.8.0).*

The layer should fail (cause a parse error) if a line longer than 998 characters is encountered or constructed after unfolding. When unparsing, if a line longer than 998 cannot be folded due to no spaces/tabs being present in it, then it is an unparse error.

Note that i/vCalendar, vCard, and MIME payloads held by IMF do not run into the IMF line length issues, in that they have their own line length limits that are smaller than those of IMF, and which do not require accommodation by having pre-existing spaces/tabs in the data. So such data will always be short enough lines.

6.6.3 Layer Name: lineFolded_iCalendar

For iCalendar (including vCard and vCalendar), the maximum is 75 bytes plus the CRLF, for a total of 77. Folding is inserted by inserting CRLF + a space or tab. The CRLF and the following space or tab are removed to unfold. If data happened to contain a CRLF followed by a space or tab initially, then that will be lost when the data is parsed.

For MIME, the maximum line length is 76.