```
    <w:lsdException w:name="heading 6" w:priority="1" w:qformat="on"/>
    <w:lsdException w:name="heading 7" w:priority="1" w:qformat="on"/>
    <w:lsdException w:name="heading 8" w:priority="1" w:qformat="on"/>
    <w:lsdException w:name="heading 9" w:priority="1" w:qformat="on"/>
    <w:lsdException w:name="Normal Indent" w:priority="6" w:qformat="on"/>
  </w:latentStyles>
```

The attributes on the latentStyles element define the properties applied to all latent styles for this document. All styles whose properties do not match the default latent styles properties are explicitly defined using the values on the lsdException elements.

2.9 Fonts

2.9.1 Font References

Within a WordprocessingML document, font face information can be referenced by any set of run properties, both as part of a style definition or direct formatting on one or more runs in the document's contents. This reference is established by referencing the primary name of the font face that is used in the rFonts element of the run properties, linking that run with the desired font face.

For example, consider a run of text that has been directly formatted to use the `Arial Black` font face. This setting would be specified as follows on the run's properties:

```
  <w:r>
    <w:rPr>
      <w:rFonts w:ascii="Arial Black" w:hAnsi="Arial Black" />
    </w:rPr>
    <w:t>This run of text uses the Arial Black font face.</w:t>
  </w:r>
```

The rFonts element specifies that the run should be formatted using the `Arial Black` font face. Applications can then look up and use the font with primary name of `Arial Black` when formatting this run.

2.9.2 Font Reference Types

In the example above, two attributes were present, both referring to the font face with primary name `Arial Black`. This simple case illustrates the ability for a WordprocessingML document to store up to four fonts which may be used on the contents of a run, as follows:

- ASCII font
- High ANSI font
- East Asian font
- Complex Script font

Each of these font faces is used to format the characters in the run that fall under their purview:

The *ASCII font* formats all characters in the ASCII range (character values 0–127). This font is specified using the ascii attribute on the rFonts element.

The *East Asian font* formats all characters that belong to Unicode sub ranges for East Asian languages. This font is specified using the eastAsia attribute on the rFonts element.

The *complex script font* formats all characters that belong to Unicode sub ranges for complex script languages. This font is specified using the cs attribute on the rFonts element.

The *high ANSI font* formats all characters that belong to Unicode sub ranges other than those explicitly included by one of the groups above. This font is specified using the hAnsi attribute on the rFonts element.

For example, consider a run of text defined as follows:

```
<w:r>
  <w:rPr>
    <w:rFonts w:ascii="Arial Black" w:hAnsi="Arial Black" w:cs="Arial"
      w:eastAsia="SimSun"/>
  </w:rPr>
  …
</w:r>
```

The rFonts element specifies that the contents of this run are formatted as follows:

- Complex script characters used the `Arial` font
- East Asian characters used the `SimSun` font
- All other characters used the `Arial Black` font

2.9.3 Ambiguous Characters

When classifying characters into one of the four slots defined above, it is likely that the classification of some characters will be ambiguous (the resulting classification would be equally applicable for one or more font slots).

To handle this, the font face information can also include a hint, which specifies how ambiguous mappings are resolved into a font slot. This information is stored on the hint attribute on the rFonts element, and specifies the bucket into which these ambiguous characters fall.

For example, if the hint attribute has a value of `eastAsia`, then all ambiguous characters shall be formatted using the East Asian font face.

2.9.4 Font Table

Within a document, the *font table* contains information about the fonts used in the document to allow:

- Applications to perform substitution with the most appropriate possible font when the desired font face is not available on the system. Since some fonts are commercially distributed, it is possible for a

document to be formatted with one or more fonts that are not available depending on the machine opening the current system. This information allows the application that cannot locate the desired font to perform the most appropriate possible match.

- Embedding of fonts in the document to prevent the need for font substitution

The font table part is stored via an implicit relationship from either the main document part or the glossary document part, and has a relationship type of http://schemas.openxmlformats.org/wordprocessingml/2006/fontTable, and a content type of vnd-openxmlformats.officedocument.wordprocessingml-fontTable+xml.

2.9.5 Font Substitution Data

The first classification of data stored in the font table are an optional set of font metrics which are queried from the font and stored in the document such that future applications can utilize them when the desired font is not available. If a particular font face cannot be located on the current system, then this data is used to substitute a font that most appropriately matches its characteristics.

For example, consider the font substitution data stored for the `Arial Black` font:

```
<w:font w:name="Arial Black">
  <w:panose1 w:val="020B0A04020102020204" />
  <w:charset w:val="00" />
  <w:family w:val="swiss" />
  <w:pitch w:val="variable" />
  <w:sig w:usb0="00000287" w:usb1="00000000" w:usb2="00000000"
    w:usb3="00000000" w:csb0="0000009F" w:csb1="00000000" />
</w:font>
```

This data is linked to the font face with a name of `Arial Black` via the name attribute, and stores the following information about the font (see the reference material on fonts for more details):

- The font's Panose-1 number
- The character set of the font
- The font's family
- The font's pitch
- The code pages and Unicode sub ranges supported by the font

2.9.6 Font Embedding

As well as providing information about the font's metrics, applications may be directed to embed the contents of a font (partially or as a whole) into a document, a process known as font embedding. *Font embedding* literally embeds an obfuscated version of the font into the file so that it may be retrieved and used to view the contents of this document - but the obfuscation ensures that the font cannot be extracted and used for any other document (as it may have a commercial license).

Within the font table, when a font is embedded there are explicit relationships to each font form needed:

- Regular
- Bold
- Italic
- Bold + Italic

Each form is obfuscated using the mechanism described in the reference material on this subject.

2.9.7 Theme Fonts

As well as storing standard font face information, run properties may store an abstraction for font face information known as theme fonts. *Theme fonts* are values that specify that the font face information for a run is not stored in the attribute value using the appropriate font face name, but is rather a reference into the document's theme part, allowing font face information to be stored and managed centrally as part of the theme data. It is appropriate to think of theme fonts as a "style for fonts" in the same way in which a style is a reference to the formatting that is stored centrally in another part.

Theme fonts are specified using the theme attribute variants in the rFonts element, rather than storing the actual font face name.

For example, consider a run of text defined as follows:

```
<w:r>
  <w:rPr>
    <w:rFonts w:asciiTheme="minorHAnsi" w:hAnsiTheme="minorHAnsi" />
  </w:rPr>
  …
</w:r>
```

The rFonts element's attribute values of asciiTheme and hAnsiTheme both store a reference to a theme font stored in the document's theme part (i.e., there is no font with the primary name `minorHAnsi`).

Once this information has been established, it is combined with the theme language data stored in the document's settings to resolve the appropriate theme fonts from the theme part. The syntax and format of the theme part are stored in the DrawingML syntax and discussed in that section.

2.10 Numbering

Numbering in WordprocessingML refers to symbols—Arabic numerals, Roman numerals, symbol characters ("bullets"), text strings, etc.—that are used to label individual paragraphs of text.

The following two paragraphs each contain numbering as defined by WordprocessingML: the first uses an Arabic numeral, the second a symbol character:

1. This is a paragraph with numbering information.

- This is also a paragraph with numbering information.

```
<additionalCharacteristics>
  <characteristic name='precisionMantissa'
    relation='gt'
    val='-9007199254740992'/>
  <characteristic name='precisionMantissa'
    relation='lt' val='9007199254740992'/>
  <characteristic name='precisionExponent'
    relation='ge' val='-1075'/>
  <characteristic name='precisionExponent'
    relation='le' val='970'/>
</additionalCharacteristics>
```

8.2 Embeddings

Office Open XML provides facilities allowing the embedding of any object within a document. For example, a WordprocessingML document might include data as an embedded SpreadsheetML document rather than a native WordprocessingML table, in order to allow that data to be edited and recalculated by a SpreadsheetML calculation engine, rather than having it stored as a static table of data.

Office Open XML provides for two classes of embedded objects:

- *Embedded Packages* - An embedded Office Open XML document embedded within another Office Open XML document, with both documents stored in the format defined by this Office Open XML specification. For example, a PresentationML document embedded within a SpreadsheetML document results in an embedded package.
- *Embedded Objects* - Any other embedded object data. The data stored in the object shall be identified by a unique string, referred to as its *ProgID*. This string shall be used to determine both the type of data and the application (if any) that shall be used to load and edit the embedded object data.

Office Open XML also allows an image to be optionally associated with the embedded object data, for use when the embedded object application and data itself is not used by the consuming application (e.g. when the object cannot be loaded – the object is from an unknown source; the object is known, but the application has chosen not to load it for performance reasons, and so on).

8.2.1 Embedded Packages

Whenever an Office Open XML document is stored as an embedded object, the embedding shall be referred to as an embedded package. Embedded packages shall be the target of the Embedded Package relationship defined in Part 1: http://schemas.openxmlformats.org/officeDocument/2006/relationships/package. this Office Open XML specification

8.2.2 Embedded Objects

For all other embeddings, the embedded object is stored in an arbitrary format defined by the application whose data is being embedded. These generic embedded objects shall be the target of the Embedded Object relationship: http://schemas.openxmlformats.org/officeDocument/2006/relationships/oleObject. When

parsing the data stored in an embedded object part, an application shall use the associated ProgID (whose location is described in the following subclauses) for the object.

8.2.3 Embeddings in a WordprocessingML Document

When an embedding is stored in a WordprocessingML document, it is stored in one of the following ways:

- In line with text - The object is displayed within the regular text stream (modifying line height and so on to accommodate it).
- Floating – The object is positioned absolutely or relatively within the document and text flow is modified as needed around it.

Each case permits the storage of both the object and the optional VML representation of the image that may be used when the object data is not used by the hosting application as follows:

8.2.3.1 Embeddings In Line With Text

When the embedding is present in line with text, it is stored as follows:

- The WordprocessingML object element specifies the presence of an embedded object in line with text.
- The child Office VML Drawing OLEObject element shall specify the details about the embedding itself, including an explicit relationship to the appropriate Embedded Package or Embedded Object part.
- The child VML shape element shall specify the presence of the image which may be used to represent the object.

For example, if we embed a SpreadsheetML worksheet in a WordprocessingML document, the following run content would be present:

```
<w:r>
  <w:object w:dxaOrig="7247" w:dyaOrig="2920">
    <v:shape id="_x0000_i1026" type="#_x0000_t75"
      style="width:362.25pt;height:146.25pt" o:ole="">
      <v:imagedata r:id="rId6" o:title="" />
    </v:shape>
    <o:OLEObject Type="Embed" ProgID="Excel.Sheet.8"
      ShapeID="_x0000_i1026" DrawAspect="Content" ObjectID="_1218026609"
      r:id="rId7" />
  </w:object>
</w:r>
```

If we examine this markup, it can be seen that:

- We have an inline embedded object, as defined by the object element.
- The OLEObject element specifies that that object is stored as an `Embed`, and that its ProgID is `Excel.Sheet.8` (the ProgID code for Microsoft Excel worksheets); it also specifies that the associated

image (when the object data cannot be used) is stored in the VML shape with a shape ID of `_x0000_i1026`.

- The associated VML shape element with an id attribute value of `_x0000_i1026` shall be used in place of the object whenever it is not loaded - this shape is typically, but is not required to be, stored in the same object element as the OLEObject element. This shape specifies its desired size and provides an explicit relationship to the part that stores the image data.

8.2.3.2 Floating Embeddings

When the embedding is present as a floating object, it is stored as follows:

- The WordprocessingML pict element specifies the presence of a floating image in the document.
- The child Office VML Drawing OLEObject element shall specify the details about the embedding itself, including an explicit relationship to the appropriate Embedded Package or Embedded Object part.
- The child VML shape element shall specify the presence of the image that may be used to represent the object in place of loading the actual object data.

For example, if we embed a SpreadsheetML worksheet in a WordprocessingML document as a floating object, the following run content would be present:

```
<w:r>
  <w:pict>
    <v:shapetype id="_x0000_t75" coordsize="21600,21600" o:spt="75"
      o:preferrelative="t" path="m@4@5l@4@11@9@11@9@5xe" filled="f"
      stroked="f">
      …
    </v:shapetype>
    <v:shape id="_x0000_s1028" type="#_x0000_t75"
      style="position:absolute;margin-left:354.75pt;margin-
      top:642.75pt;width:182.3pt;height:73.6pt;z-index:251660288">
     <v:imagedata r:id="rId4" o:title="" />
    </v:shape>
    <o:OLEObject Type="Embed" ProgID="Excel.Sheet.8"
      ShapeID="_x0000_s1028" DrawAspect="Content" ObjectID="_1218026611"
      r:id="rId5" />
  </w:pict>
</w:r>
```

If we examine this markup, it can be seen that:

- We have a floating image, as defined by the pict element.
- The OLEObject element specifies that that floating image is actually an embedding that is stored as an Embed, and that its ProgID is `Excel.Sheet.8` (the ProgID code for Microsoft Excel worksheets); it also specifies that the associated image (when the object data cannot be used) is stored in the VML shape with a shape ID of `_x0000_s1028`.

- The associated VML shape element with an id attribute value of `_x0000_s1028` shall be used in place of the object whenever it is not loaded - this shape is typically, but is not required to be, stored in the same pict element as the corresponding OLEObject element. This shape specifies its desired size and provides an explicit relationship to the part which stores the image data.

8.2.4 Embeddings in a SpreadsheetML Document

When an embedding is present in a SpreadsheetML document, it shall be stored as follows:

- In the worksheet, the oleObjects element shall store one or more oleObject child elements, one for each embedding within the current worksheet. Each of those oleObject child elements shall also store: an explicit relationship to the associated Embedded Package or Embedded Object part, the ProgID for that embedded object, and (optionally) the last four digits of the shape ID for the associated VML shape. The shape ID itself shall be of the form `_x0000_s####`, where # specifies a single Arabic numeral, in order to be referenced as the alternate image for an embedding in a SpreadsheetML document.
- In the worksheet, the sibling legacyDrawing element shall contain an explicit relationship to the VML Drawing part that (optionally) contains the image data which may be used in place of loading the actual object data.

For example, if we embed a `Contoso Test` object (an example for illustration) in a SpreadsheetML document, the following markup would be stored in the appropriate Sheet part:

```
<s:worksheet>
  …
  <s:legacyDrawing r:id="rId9" />
  <s:oleObjects>
    <s:oleObject progId="Contoso.Test.1" shapeId="1025" r:id="rId5"/>
  </s:oleObjects>
</s:worksheet>
```

If we examine this markup, it can be seen that:

- The oleObject element specifies that we have one embedded object on the worksheet. Its attributes specify that the object is of type `Contoso.Test.1` and that the explicit relationship to the embedded object is `rId5`.
- The sibling legacyDrawing element specifies that the Legacy Drawing part which contains the associated legacy drawing data is contained at the target of the relationship with an ID of `rId9`.
- If we examine the VML Drawing part's contents, we'll see the shape which ends in `1025`, which contains the alternate image for the object:

```
<v:shape id="_x0000_s1025" type="#_x0000_t75" style='position:absolute;
  margin-left:240.75pt;margin-top:105.75pt;width:334.5pt;height:253.5pt;
  z-index:1' filled="t" fillcolor="window [65]" stroked="t"
  strokecolor="windowText [64]" o:insetmode="auto">
  <v:fill color2="window [65]"/>
  <v:imagedata o:relid="rId1" o:title=""/>
  <x:ClientData ObjectType="Pict">
    <x:SizeWithCells/>
    <x:Anchor>5, 1, 7, 1, 11, 63, 23, 19</x:Anchor>
    <x:CF>Pict</x:CF>
  </x:ClientData>
</v:shape>
```

8.2.5 Embeddings in a PresentationML Document

When an embedding is present in a PresentationML document, it shall be stored as follows:

- In the slide, the embedding is stored as a graphic frame using the graphicFrame element in PresentationML.
- The graphicData element for the frame shall have the appropriate URI for its contents: http://schemas.openxmlformats.org/presentationml/2006/ole. Its child element shall be the PresentationML oleObj element, which stores an explicit relationship to the associated Embedded Package or Embedded Object part, the ProgID for that embedded object, and (optionally) the shape ID for the associated VML shape.
- The Slide part shall also have an implicit relationship to a VML Drawing part that (optionally) contains the image data to be used in place of loading the actual object data.

For example, if we embed the `Equation.3` object in a PresentationML document, the following markup would be stored in the shape tree of the appropriate Slide part:

```
<p:graphicFrame>
  …
  <a:graphic>
  C:\Documents and Settings\tristand\Local Settings\Temp\Temporary Directory 4 for
embeddedObject.pptx.zip\ppt\slides\slide1.xml   <a:graphicData
      uri="http://schemas.openxmlformats.org/presentationml/2006/ole">
      <p:oleObj spid="_x0000_s1026" name="Equation" r:id="rId3"
        imgW="320" imgH="272" progId="Equation.3">
        <p:embed />
      </p:oleObj>
    </a:graphicData>
  </a:graphic>
</p:graphicFrame>
```

If we examine this markup, it can be seen that:

- The uri attribute on the graphicData element is http://schemas.openxmlformats.org/presentationml/2006/ole, which dictates that this is an embedded object
- It contains an oleObj element that specifies that the properties of the embedded object. Its attributes specify that the object is of type `Equation.3` and that the explicit relationship to the embedded object is `rId3`.
- The slide may also contain an implicit relationship to a Legacy Drawing part. If we examine the legacy drawing part's contents, the shape with ID `_x0000_s1026` (if present) defines the alternate image:

```
<v:shape id="_x0000_s1026" type="#_x0000_t75" style='position:absolute;
  left:282pt;top:24pt;width:152pt;height:129.25pt'>
  <v:imagedata o:relid="rId1" o:title=""/>
</v:shape>
```

8.3 Future Extensibility

This clause provides a high-level overview of the extensibility model for Office Open XML documents, and a description of packaging conventions in the context of DrawingML and PresentationML. Two main constructs are described: extensibility lists (extLst/ext) and alternate content blocks (AlternateContent).

To illustrate certain points, a number of examples refer to versions of a (fictitious) PresentationML consumer/producer called PML. The 2003 version is called PML 2003; the 2007 version is called PML 2007; and so on.

8.3.1 Terminology

Here are some terms useful when discussing future extensibility.

- *Round tripping* involves the interchange of documents between different consumers/producers, as well as between different versions of the same consumer/producer. The pair of consumers/producers can be on the same or different platforms. Consider the case in which a document is created by the PML 2007. This document is then opened by PML 2003, edited, and saved. The edited document is now opened and used by PML 2007. In this case, the document originally created by PML 2007 has been round-tripped through PML 2003. It is also possible to round-trip a document created by PML 2003 through PML 2007.
- A *Downrev* (or down-level) version of a consumer/producer refers to one that understands an older version of a given schema. An *Uprev* (or up-level) version of a consumer/producer refers to one that understands a newer version of a given schema. The terms Downrev and Uprev are typically used in relative reference to one another. As an example, let's consider again, the two consumer/producer PML 2003 and PML 2007, where PML 2007 was released sometime after PML 2003, and, consequently, PML 2007 understands a newer revision of the DrawingML schema then does PML 2003. PML 2003 is referred to as the Downrev version while PML 2007 is referred to as the Uprev version. It is assumed that the Downrev version has less capability than the Uprev version.