Extension Mechanisms Best Practises [informative section]

There are two broad ways to add extra data to documents – using the extension mechanisms described in Part 3 (Markup Compatibility and Extensibility) and embedding foreign OPC parts. MCE offers three primary mechanisms for extending XML files, each with its own advantages and disadvantages.

**MCE: Ignorable elements and attributes (§xxx)**  
The most commonly-used extension mechanism, marking elements or attributes as ignorable allows lightweight additions to be made to existing markup.

A good use of ignorable markup would be the addition of a custom metadata tag onto a paragraph in a WordprocessingML document. This could be accomplished by declaring a custom namespace, marking it as ignorable and adding the attribute to the p element in that namespace. The relevant portions of resulting document.xml part might resemble the below:

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

<w:document xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" xmlns:mymeta="http://mywordprocessorapp.com/metadata" mc:Ignorable="mymeta">

<w:body>

<w:p mymeta:tag="marketing\_team" w:rsidR="00AD3E96" w:rsidRDefault="00120C37">

<w:r>

<w:t>hello</w:t>

</w:r>

</w:p>

</w:body>

</w:document>

Ignorable markup can be used anywhere in XML parts and requires minimal markup. It allows custom markup to be added to documents while retaining the document's conformance with the standard and allowing it to be opened in a third party application without errors. However, ignorable elements and attributes will almost definitely be lost if files are "round tripped" (opened and saved again) in an application which does not understand them, as there is no requirement for applications to persist ignorable markup, and typically unknown ignorable markup is stripped during file load.

**MCE: Alternate Content Blocks (§xxx)**

While ignorable constructs allow markup to easily be added to documents, Alternate Content Blocks (ACBs) allow existing markup to be replaced, with the replacement targeted at particular consumers which understand it.

A good use of ACBs would be in developing an application which preferred to use the ODF format in WordprocessingML paragraph markup. When creating files, our application would continue to write OpenXML markup in order to be compliant to the standard, but would also provide ODF markup in an ACB. When opening files, our application would disregard the OpenXML fallback markup and only read the ODF.

The resulting document.xml part for a document with text stored in such a way might appear as below (not that, for simplicity, all ODF namespaces are merged into one):

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

<w:document xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" xmlns:myodf="http://mywordprocessorapp.com/odfcontent" mc:Ignorable="myodf">

<w:body>

<w:p w:rsidR="00AD3E96" w:rsidRDefault="00120C37">

<mc:AlternateContent>

<mc:Choice Requires="myodf">

<myodf:style myodf:name="T2" myodf:parent-style-name="DefaultParagraphFont" myodf:family="text">

<myodf:text-properties myodf:font-style="italic" myodf:font-style-asian="italic"/>

</myodf:style>

<myodf:p>This document is stored in<myodf:s/>

<myodf:span myodf:style-name="T2">two</myodf:span>

<myodf:s/>formats.</myodf:p>

</mc:Choice>

<mc:Fallback>

<w:r>

<w:t xml:space="preserve">This document is stored in </w:t>

</w:r>

<w:r>

<w:rPr>

<w:i/>

</w:rPr>

<w:t>two</w:t>

</w:r>

<w:r>

<w:t xml:space="preserve"> formats.</w:t>

</w:r>

</mc:Fallback>

</mc:AlternateContent>

</w:p>

</w:body>

</w:document>

ACBs allow for the replacement of existing markup for consumers which understand it. Much like ignorables, there is no requirement for applications to preserve ACBs on round-trip, so data may be lost if third-party applications are used to open and save files.

**MCE: Application-defined extension elements (§xxx)**

Application-defined extension elements essentially allow markup designers to put "this space left for future expansion" elements into their formats. Syntactically, these are similarly to ignorable elements but, because they only appear at predefined locations, markup consumers can easily keep track of unknown extension elements which makes round-tripping a simpler proposition.

In ISO/IEC29500-1, SpreadsheetML is the only one of the three markups which utilises extension elements – it uses the extLst element (Part 1, 18.2.10). extLst elements occur at several predefined points in SpreadsheetML and allow the markup to be extended in a manner which permits round-trip.

A good use of SpreadsheetML's extension elements would be a spreadsheet application whose developers wished to add the ability for cells to be denoted as model inputs or outputs. The application in question could use these tags at runtime and, if users were to round-trip the files in other applications, the markup would be preserved.

The CT\_Cell type in SpreadsheetML contains an exLst element, so this will be an acceptable extension point. It contains an unbounded collection of ext elements, and the developer may add an ext with their extension's markup.The resulting sheetdata for a given spreadsheet might therefore appear as below:

<sheetData>

<row r="1" spans="1:1" x14ac:dyDescent="0.25">

<c r="A1">

<v>1234</v>

<extLst>

<ext xmlns:mymodel="http://myspreadsheetapp.com/modelInputsAndOutputs" uri="http://myspreadsheetapp.com/modelInputsAndOutputs">

<mymodel:modelSpecifier cellType="input" name="interestRate">

<mymodel:description>Specifies the interest rate to be passed into amortisation model.</mymodel:description>

</mymodel:modelSpecifier>

</ext>

</extLst>

</c>

</row>

</sheetData>

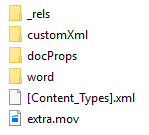
Because consuming spreadsheet applications understand that this data is attached at a cell level, this metadata will remain with the cell when it is moved around the sheet via cut/paste or through row/column insertion or deletions above it. Note that some implementations may parse through application-defined-extension elements and modify constructs within them – Microsoft Excel, for example, will look for any sqref elements in the namespace <http://schemas.microsoft.com/office/excel/2006/main>. It assumes that they will contain spreadsheet row/column references and adjust them appropriately if that referenced cell area is moved around at runtime.

Application-defined extension elements are only usable in locations pre-defined by a markup language, but allow for data preservation in round-trip scenarios.

**Foreign OPC parts**

Although not an MCE construct, using of foreign OPC parts is a valid and useful means of storing extra data inside documents. Markup consumers are able (but not required) to preserve these parts during save operations (Part 1, 9.1.4). Unknown parts are best suited to large amounts of data (either binary or XML) which the creator desires to be preserved on round-trip.

A good use of unknown parts would be for an embedded video file attached to a WordprocessingML document. First, we need to add the file to the OPC package:



And then add a relationship to the \\_rels\.rels part:

<Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships">

<Relationship Id="rId5" Type="http://example.org/myexample" Target="extra.mov"/>

</Relationships>

The content will likely be preserved on round-trip through non-understanding applications, and because there’s no requirement to serialise it into XML this extension mechanism is well-suited to binary data such as video or images.