XMLTEX: A non validating (and not 100% conforming) namespace aware XML parser implemented in TEX

David Carlisle
NAG Ltd
Jordan Hill Road
Oxford
davidc@nag.co.uk

Abstract

XMLTEX implements a non validating parser for documents matching the W3C XML Namespaces Recommendation.

Introduction

XMLTEX may be used simply to parse a file (expanding entity references and normalising namespace declarations) in which case it records a trace of the parse on the terminal. However, in normal use, the information from the parse is used to trigger TEX typesetting code. Declarations (in TEX syntax) are provided as part of XMLTEX to associate TEX code with the start and end of each XML element, attributes, processing instructions, and with unicode character data.

Installation

The XMLTEX parser itself does not require LATEX. It may be loaded into `initex` to produce a format capable of parsing XML files. However such a format would have no convenient commands for typesetting, and so normally XMLTEX will be used on top of an existing format, normally LATEX. In this section we assume that the document to be processed is called `document.xml`.

Using XMLTEX as an input to the LATEX command LATEX requires a document in TEX syntax, not XML. To process `document.xml`, first produce a two line file called `document.tex` of the following form:

```
\def\xmlfile{document.xml}
\input xmltex.tex
```

No other commands should appear in this file!

The document may then be processed with the command: `latex document`, or some equivalent procedure in the user's TEX environment.

Using XMLTEX as a TEX format built on LATEX Some users may prefer to set up XMLTEX as a format in its own right. This may speed things up slightly (as `xmltex.tex` need not be read each time) but more importantly perhaps it allows the XML file to be processed directly without needing to make the `.tex` wrapper.

To make a format, some command such as the following is used, depending on the user's TEX system.

```
initex &latex xmltex
initex \&latex xmltex
tex -ini &latex xmltex
tex -ini \&latex xmltex
```

This will produce a format file `xmltex.fmt`. It is then possible to make an `xmltex` command by copying the way the `latex` command is defined in terms of `latex.fmt`. Depending on the TEX system, this might be a symbolic link, or a shell script, or batch file, or a configuration option in a setup menu.

Making an XMLTEX format 'from scratch' It may be convenient, for some, to build an XMLTEX format as above, starting from the LATEX format. However, other users may prefer to work with an `initex` with no existing format file. Even for those who wish to use standard LATEX it may be preferable to make a TEX input file that first inputs `latex.ltx` then `xmltex.tex`. In particular this permits different hyphenation and language customisation for XMLTEX than for LATEX. Many of the features of the language support in LATEX are related to modifying the input syntax to be more convenient. Such changes are not needed in XMLTEX as the input syntax is always XML. Some language files may change the meaning of such characters as `<` which would break the XMLTEX parser. Also, rather than using `latex.ltx` one can in principle use a modified `docstrip` install file and produce a 'cut down' LATEX that omits features that are not going to be used in XMLTEX.

Unfortunately the support for this method of building XMLTEX (and access to non-English hyphenation generally) is not fully designed and totally undocumented.

Using XMLTEX

XMLTEX by default 'knows' nothing about any particular type of XML file, and so needs to load external files containing specific information. This section describes how the information in the XML file determines which files will be loaded.

1. If the file begins with a Byte Order Mark, the default encoding is set to UTF-16. Otherwise the default encoding is UTF-8.
2. If (after an optional BOM) the document begins with an XML declaration that specifies an encoding, this encoding will be used, otherwise the default encoding will be used. A file with name of the form *encoding*`.xmt` will be loaded that maps the requested encoding to Unicode positions. (It is an error if this file does not exist for the requested encoding.)
3. If the document has a DOCTYPE declaration that includes a local subset then this will be parsed. If any external DTD entity is referenced (by declaring and then referencing a parameter entity) then the SYSTEM and PUBLIC identifiers of this entity will be looked up in a catalogue (to be described below). If either identifier is known in the catalogue the corresponding XMLTEX package (often with `.xmt` extension) will be loaded.
4. After any local subset has been processed, if the DOCTYPE specifies an external entity, the PUBLIC and/or SYSTEM identifiers of the external DTD file will be similarly looked up, and a corresponding XMLTEX file loaded if known.
5. As each element is processed, it may be 'known' to XMLTEX by virtue of one of the packages loaded, or it may be unknown. If it is unknown then if it is in a declared namespace, the namespace URI (not the prefix) is looked up in the XMLTEX catalogue. If the catalogue specifies an XMLTEX package for this namespace it will be loaded. If the element is not in a namespace, then the element name will be looked up in the catalogue.
6. If after all these steps the element is still unknown then depending on the configuration setting either a warning or an error will be displayed. (Currently only warning implemented.)

The XMLTEX Catalogue As has already been explained, XMLTEX requires a mapping between PUBLIC and SYSTEM identifiers, namespace URI, and element names, to files of TEX code. This mapping is implemented by the following commands:
`\NAMESPACE{`*URI*`}{`*xmt-file*`}`
`\PUBLIC{`*FPI*`}{`*file*`}`
`\SYSTEM{`*URI*`}{`*file*`}`
`\NAME{`*element-name*`}{`*xmt-file*`}`
`\XMLNS{`*element-name*`}{`*URI*`}`

As described above, if the first argument of one of these commands matches the string specified in the XML source file, the corresponding TEX commands in the file specified in the second argument are loaded. The PUBLIC and SYSTEM catalogue entries may also be used to control which XML files should be input in response to external entity references. The `\XMLNS` command is rather different; if an element in the null namespace does not have any definition attatched to it, this declaration forces the default namespace to the given URI. The catalogue lookup is then repeated. This allows for example documents beginning `<html>` to be coerced into the XHTML namespace.

These commands may be placed in a configuration file, either `xmltex.cfg`, in which case they apply to all documents, or in a configuration file '`\jobname.cfg`' (e.g., `document.cfg` in the example in section 'Using XMLTEX', above) in which case the commands just apply to the specified document.

Configuring XMLTEX In addition to the 'catalogue' commands described earlier there are other commands that may be placed in the configuration files.

- `\xmltraceonly`
  This stops XMLTEX from trying to typeset the document. The external files specified in the catalogue are still loaded, so that the trace may report any elements for which no code is defined, but no actual typesetting takes place. In the event of unknown errors it is always worth using XMLTEX in this mode to isolate any problems.

  It may be noted that if an XMLTEX format is built just using `initex` without any typesetting commands, the resulting format should still be able to parse any XML file if `xmltex.cfg` just specifies `\xmltraceonly` and `\jobname.cfg` is empty.
- `\xmltraceoff`
  By default XMLTEX provides a trace of its XML parse, displaying each element begin and end.

This command in `xmltex.cfg` or `\jobname.cfg` will stop the trace being produced.

- `\inputonce{`*xmt-file*`}`
  The catalogue entries specify that certain files should be loaded if XML constructs are met. Alternatively the files may just always be loaded. The system will ignore any later requests to load. This is especially useful if an XMLTEX format is being made.

- `\UnicodeCharacter{`*hex-or-dec*`}{`*tex-code*`}`
  The first argument specifies a unicode character number, in the same format as used for XML character entities, namely either a decimal number, or an upper case Hex number preceded by a lower case '`x`'.

  The second argument specifies arbitrary TEX code to be used when typesetting this character. Any code in the XML range may be specified (i.e., up to x10FFFF). Although codes in the 'ASCII' range, below 128, may be specified, the definitions supplied for such characters will not by default be used. The definition will however be stored and used if the character is activated using the command described below.

- `\ActivateASCII{`*hex-or-dec*`}`
  The argument to this command should be a number less than 128. If a character is activated by this command in a configuration file then any special typesetting instructions specified for the character will be executed whenever the character appears as character data.

  Some ASCII characters are activated by default. The list is essentially those characters with special meanings to either TEX or XML.

If a format is being made, there are essentially two copies of `xmltex.cfg` that may play a role. The configuration file input when the format is made will control catalogue entries and packages built into the format. A possibly different `xmltex.cfg` may be used in the input path of 'normal' TEX, this will then be used for additional information loaded each run.

In either case, a separate configuration file specific to the given XML document may also be used (which is loaded immediately after `xmltex.cfg`).

Stopping xmltex

XMLTEX should stop after the end of the document element has been processed. If something goes wrong one may be offered TEX's `*` prompt from which one might choose to exit with `<?xmltex stop?>`.

XMLTEX package files

XMLTEX package files are the link between the XML markup and TEX typesetting code. They are written in TEX (rather than XML) syntax and may load directly or indirectly other files, including LATEX class and package files. For example a file loaded for a particular document type may directly execute `\LoadClass{article}`, or alternatively it may cause some XML element in the document to execute `\documentclass{article}`. In either case the document will suffer the dubious benefit of being formatted according to the style implemented in the standard article class. Beware though that the package files may be loaded at strange times, the first time a given namespace is declared in a document, and so the code should be written to work if loaded inside a local group.

Characters in XMLTEX package files have their normal LATEX meanings except that line endings are ignored so that there is no need to add a `%` to the end of lines in macro code. Unlike LATEX `.fd` file conventions, other white space is *not* ignored.

The available commands are:

- `\FileEncoding{`*encoding*`}`
  This is the analogue for TEX syntax files of the encoding specification in the XML or text declaration of XML files. If it is not specified the file will be assumed to be in UTF-8.

- `\DeclareNamespace{`*prefix*`}{`*URI*`}`
  This declares a prefix to be used *in this file* for referring to elements in the specified namespace. If the prefix is empty then this declares the default namespace (otherwise, unprefixed element names refer to elements that are not in a namespace).

  Note that the elements in the XML document instance may use a different prefix, or no prefix at all to access this namespace. In order to resolve these different prefixes for the same namespace, each time a namespace is encountered for the first time (either by `\DeclareNamespace` in a preloaded package, or in a namespace declaration in the XML instance) then it is allocated a new number and any further namespace declaration for the same URI just locally associates a prefix with this number. It is these numbers that are displayed when the XML trace of the parse of the document is shown, and also if any element is written out to an external file it will have a normalised numerical prefix whichever prefix it originally had. (Numeric prefixes are not legal XML, but this is an advantage, as it

ensures these internal forms can not clash with any prefix actually used in the document.)

Three namespaces are predeclared. The null namespace (0), the XML namespace `http://www.w3.org/1998/xml` (1) which is predeclared with prefix `xml` as specified in the Namespace Recommendation, and the XMLTEX namespace `http://www.dcarlisle.demon.co.uk/xmltex` (2) which is not given a default prefix, but may be used to have XML syntax for some internal commands (eg to have .aux files fully in XML, currently they are a hybrid mixture of some TEX and some XML syntax).

- `\XMLelement{`*element-qname*`}{`*attribute-spec*`}{`*begin-code*`}{`*end-code*`}`
  This is similar to a LATEX `\newenvironment` command. It declares the code to execute at the start and end of each instance of this element type. This code will be executed in a local group (like a LATEX environment). The second argument declares a list of attributes and their default values using the `\XMLattribute` command whose description follows.
- `\XMLelement{`*element-qname*`}{`*attribute-spec*`}{\xmlgrab}{`*end-code*`}`
  A special case of the above command (which may be better made into a separate declaration) is to make the *start-code* just be the command `\xmlgrab`. In this case the *end-code* has access to the element content (in XML syntax) as `#1`. This content isn't literally the same as the original document, namespaces, white space and attribute quote symbols will all have been normalised.
- `\XMLattribute{`*attribute-qname*`}{`*command-name*`}{`*default*`}`
  This command may only be used in the argument to `\XMLelement`. The first argument specifies the name of an attribute (using any namespace prefixes current for this package file, which need not be the same as the prefixes used in the document). The second argument gives a TEX command name that will be used to access the value of this attribute in the begin and end code for the element. (Note using TEX syntax here provides a name independednt of the namespace declarations that are in scope when this code is executed). The third argument provides a default value that wil be used if the attribute is not used on an instance of this element.

  The special token `\inherit` will cause the command to have a value set in an ancestor element if this element does not specify any value.

  If a TEX token such as `\relax` is used as the default the element code may distinguish the case that the attribute is not used in the document.
- `\XMLnamespaceattribute{`*prefix*`}{`*attribute-qname*`}{`*command-name*`}{`*default*`}`
  This command is similar to `\XMLattribute` but is used at the top level of the package file, not in the argument to `\XMLelement`. It is equivalent to specifying the attribute in *every* element in the namespace specified by the first argument. As usual the prefix (which may be to denote the default namespace) refers to the namespace declarations in the XMLTEX package: the prefixes used in the document may be different.
- `\XMLentity{`*name*`}{`*code*`}`
  Declare an (internal parsed) entity, this is equivalent to a `<!ENTITY]]>` declaration, except that the replacement text is specified in TEX syntax.
- `\XMLname{`*name*`}{`*command-name*`}`
  Declare the TEX command to hold the (normalised, internal form) of the XML name given in the first argument. This allows the code specified in `\XMLelement` to refer to XML element names without knowing the encodings or namespace prefixes used in the document. Of particular use might be to compare such a name with `\ifx\XML@parent` which will allow element code to take different actions depending on the parent of the current element.
- `\XMLstring{`*command-name*`}<>`*XML Data*`</>`
  This saves the XML fragment as the TEX command given in the first argument. It may be particularly useful for redefining 'fixed strings' that are generated by LATEX document classes to use any special typesetting rules specified for individual characters.

XML processing

XMLTEX tries as far as possible to be a fully conforming non validating parser. It fails in the following respects.

- Error reporting is virtually non existent. Names are not checked against the list of allowed characters, and various other constraints are not enforced.
- A non validating parser is not forced to read external DTD entities (and this one does not). It is obliged to read the local subset and process entity definitions and attribute declarations. Entity declarations are reasonably well handled: External parameter entities are handled as above, loading a corresponding XMLTEX

file if known. External entities are similarly processed, inputting the XML file, a difference in this case is that if the entity is not found in the catalogue, the SYSTEM identifier will be used directly to `\input` as often this is a local file reference. Internal parsed entities and parameter entities are essentially treated as TEX macros, and nonparsed entities are saved along with their NDATA type, for use presumably by `\includegraphics`.

Any default attributes specified in the local subset are saved and added to the corresponding element befor any processing is triggered. Note that this defaulting, unlike the defaults specified with `\XMLattribute` are 'namespace unaware' and only apply to elements using the same expanded name. The element from th esame namespace but represented with a different prefix will not have these defaults applied.

- Support for encodings depends on having an encoding mapping file. Any 8-bit encoding that matches Unicode for the first 127 positions may be used by making a trivial mapping file. (The one for latin1 looks over complicated as it programs a loop rather than having 127 declarations saying that latin1 and Unicode are identical in this range).

  UTF-8 is supported, but support for UTF-16 is minimal. Currently only latin-1 values work: (In this range UTF-16 is just latin-1 with a null byte inserted after (or before, depending on endedness) each latin-1 byte. The UTF-16 implementation just ignores this null byte then processes as for latin-1. Probably the first few 8-bit pages could be similarly supported by making the low ASCII control characters activate UTF-16 processing but this will never be satisfactory using a standard TEX. Hopefully a setup for a 16bit TEX such as Omega will correct this.

Accessing TEX

In theory one should be able to control the document simply by suitable code specified by `\XMLelement` and friends, but sometimes it may be necessary to 'tweak' the output by placing commands directly in the source.

Two mechanisms are availalable to do this.

- Using the XMLTEX namespace. The XMLTEX namespace conatins a small (currently empty) set of useful TEX constructs that are accessed by XML syntax. For example if XMLTEX provides a mechanism for having XML (rather than LATEX) syntax toc files, it will need an analogue of `\contentsline` which might be an element accessed by `<xmltex:contentsline>`... where the XMLTEX prefix is declared on this or a parent element to be `xmlns:xmltex="http://www.dcarlisle.demon.co.uk/xmltex"`.

  As the XMLTEX namespace is declared but currently empty, a more useful variant of this might be:

- Declare a personal namespace for TEX tweaks, and load a suitable package file that attatches TEX code to the elements in this namespace (or at least specify the correspondence between the namespace and the package using `\NAMESPACE`). For instance, `<clearpage xmlns="/my/tex/tweak"/>` will force a page break if, at suitable points, the document contains:

  ```
  \NAMESPACE{/my/tex/tweak}{tweak.xmt}
  ```

  and

  ```
  \DeclareNamespace{twk}{/my/tex/tweak}
  \XMLelement{twk:clearpage}{\clearpage}
  ```

- A second different mechanism is available, to use XML processing instructions. A Processing Instruction of the form: `?xmltex` *TEX commands* `?>` will execute the TEX commands.

Bugs

None, of course.

Don't Read Past This Point

Thus section discusses some of the more experimental features of XMLTEX that may get a cleaner syntax (or be removed, as a bad idea) in later releases, and also describes some of the internal interfaces (which are also subject to change)

Input Encodings and States At any point while processing a document, XMLTEX is in one of two *states*: *tex* or *xml*.

States In the *xml state*, `<` and `&` are the only two characters that trigger special markup codes. Other characters, such as `!`, `>`, `=`, ...; may be used in certain XML constructs as markup but unless some code has been triggered by `<` they are treated simply as character data. All characters above 127 are 'active' to TEX and are used to translate the input encoding to UTF-8. All internal character handling is based on UTF-8, as described below. Some characters in the ASCII range, below 127 are also active by default (mainly punctuation characters used in XML constructs, such as the ones listed above). Some or all of the others may be activated using the `\ActivateASCII` command, which allows special

typesetting rules to be activated for the characters, at some cost in processing speed.

In the *tex state*, characters in the ASCII range have their usual TEX meanings, so letters are 'catcode 11' and may be used in TEX control sequences, \ is the escape character, & the table cell separator, etc. Characters above 127 have the meanings current for the current encoding just as for the *xml state*, probably this means that they are unusable in TEX code, except for the special case of referring to XML element names in the first argument to `\XMLelement` and releated commands.

Encodings Whenever a new (XML or TEX) file is input by the XMLTEX system the *encoding* is first switched to UTF-8. At the end of the input the encoding is returned to whatever was the current encoding. The encoding current while the file is read is determined by the encoding pseudo-attribute on the XML or text declaration in the case of XML files, or by the `\FileEncoding` command for TEX files. Note that the encoding mechanism *only* is triggered by XMLTEX file includes. Once an XMLTEX package file is loaded it may include other TEX files by `\input` or `\includepackage` these input command swill be transparent to the XMLTEX encoding system. The vast majority of TEX macro packages only use ASCII characters so this should not be a problem.

Note that if the `\includepackage` occurs directly in the XMLTEX package file, the TEX code will be included with a known encoding, the one specified in the XMLTEX package, or UTF-8. If however the `\includepackage` is included in code specified by `\XMLelement`, then it will be executed with whatever encoding is current in the document at the point that element is reached. Before XMLTEX executes the code for that element it will switch to the *tex state*, thus normalising the ASCII characters but characters above 127 will not have predefined definitions in this case.

Internally eveything is stored as UTF-8. So `.aux` and `.toc` files will be in UTF-8 even if the document (or parts of the document) used different encodings.

To specify a new encoding, if it is an 8 bit encoding that matches ASCII in the printable ASCII range, then one just needs to produce a file with name *encoding*`.xmt` (in lowercase, on case sensitive systems) this should consist of a series of `\InputCharacter` commands, giving the input character slot and the equivalent Unicode. If an encoding is specified in this manner character data will be converted to UTF-8 by *expansion* and so ligatures and inter letter kerns will be preserved. (Conversely if characters are accessed by character references, `&#1234;` then TEX arithmetic is used to decode the information and ligature information will be lost. For some large character sets, especially for Asian languages, these mechanisms will probably not prove to be sufficient. Alternative mechanisms are being investigated, but in the short term it may be necessary to always use UTF-8 if the input encoding is not strictly a ine byte extension of the ASCII code page.

XMLTEX Package Commands And TEX command may be used in an XMLTEX package, although the user should be aware that the file may be input into a local group, at the point in a document that a particular namespace is first used, for example. There are however some specific commands designed to be used in the begin or end code of `\XMLElement`.

- `\ignorespaces`
  This is similar to the TEX primitive of the same name, but redefined to work more naturally in this context.
- `\obeyspaces`
  Obey consecutive space characters, rather than treating consecutive runs as a single space. (A command of this name, but not this definition is in plain TEX.)
- `\obeylines`
  Obey end of line characters, rather than treating then as a space, force a line break. (A command of this name, but not this definition exists in plain TEX.)
- `\xmltexfirstchild#1\@`
  If the *start-code* for an element is specified as `\xmlgrab` then the *end-code* may use `#1` in order to execute the element content. However, the entire content is not always needed, and the construction `\xmltexfirstchild\#1\@` (with currently unpleasant syntax) will just evaluate the first child element of the content, discarding the remaining elements.
- `\xmltextwochildren\csa\csb#1`
  If it is known that the content will be exactly two child elements (e.g., a MathML `frac` or `sub` element) then this command may be used. The command executes the TEX code `\csa{`*child-1*`}\csb{`*child-2*`}` So either two TEX commands may be supplied, one will be applied to each child, or the second argument may be `{}` in which case the first argument may be a TEX command that takes two arguments. For example the code for MathML frac might be

```
\XMLelement{m:mfrac}
  {}
```

```
{\xmlgrab}
{\xmltextwochildren\frac{}#1}
```

- `\xmltexthreechildren\csa\csb\csc#1`
  As above, but more so.
- `\xmltexforall\csa{#1}`
  The TEX command `\csa` is executed repeatedly, taking as argument each time one child from the content '#1' of the current element. The command name `\xml@name` is set to the (normalised, internal) name of each child element before `\csa` is executed.
- `\NDATAEntity\csa\csb\attvalue`
  If the XML parser encounters an internal or external entity reference it expands it without executing any special hook that may be defined in an XMLTEX package. However NDATA entites are never directly encountered in an entity reference. They may only be used as an attribute value. If `\attvalue` is a TEX command holding the value of an attribute, as declared in `\XMLattribute` then `\NDATAEntity\csa\csb` `\attvalue` applies the two TEX commands `\csa` and `\csb` to the notation type and the value, in a way analogous to `\xmltextwochildren`, so for example the XML version of manual document, from which this paper is derived, specifies:

```
<!NOTATION URL SYSTEM "" >
<!ENTITY lppl SYSTEM
 "http://www.latex-project.org/lppl.txt"
  NDATA URL>
```

and this is handled by the following XMLTEX code

```
\XMLelement{xptr}
 {\XMLattribute{doc}{\xptrdoc}{}}
 {\NDATAEntity\xptrdoc\@gobble\url}
 {}
```

which saves the attribute value in `\xptrdoc` and then discards the notation name (URL) and applies the command `\url` to typeset the supplied URL.

David Carlisle