DITA For Publishers User Guide

Contents

.....4
About This Book.....5
Release Notes.....6
Release Notes: Version 0.9.18.....6
Release Notes: Version 0.9.16.....8
Release Notes: Version 0.9.15.....8
Release Notes: Version 0.9.14.....9
Release Notes: Version 0.9.13.....9
Release Notes: Version 0.9.12.....9
Release Notes: Version 0.9.11.....9
Release Notes: Version 0.9.10.....10
Release Notes: Version 0.9.9.....10
Introduction to DITA For Publishers.....11
The DITA For Publishers Open Toolkit Plugins.....12
Installing the Toolkit Plugins.....12
Setting Up The kindlegen Command-Line Utility.....13
Generating EPUBs and Kindle Books from DITA Content.....13
Generating EPUBs From OxygenXML.....14
Generating Kindle Books From OxygenXML.....14
Using Custom CSS Style Sheets.....15
Creating Cover Graphics and Covers.....16
Creating Arbitrary EPUB OPF Metadata.....16
Implementing Overrides for Common Processing.....17
EPUB and Kindle Transformation Parameters.....19
HTML2 Plugin.....22
Understanding the HTML2 Transformations.....23
HTML2 Transformation Parameters.....25
Generating DITA from Documents (Word-to-DITA Transformation Framework).....26
The DITA For Publishers Markup Vocabulary.....27
Understanding the DITA For Publishers Markup Vocabulary.....27
DITA for Publishers Vocabulary Modules Overview.....29
Integrating D4P Modules Into Document Type Shells.....37
Integrating Attribute Domains.....37
DITA For Publishers Vocabulary Reference.....39
The DITA For Publishers Word to DITA and DITA to InDesign Tools.....40
The Word-to-DITA Transformation Framework.....40
Getting Started With The Word2DITA Transform.....41
Generating DITA from Word Using the Toolkit Plugin.....42
Generating DITA From Within OxygenXML.....44
Style to Tag Mapping.....45
Common Style-to-Tag Mapping Cases.....49
Troubleshooting the Word-to-DITA Process.....55
Word-to-DITA Style-to-Tag Mapping Video Tutorial.....57
Extending and Overriding the Word to DITA Transform.....57
Tips for Using Word With the Word-to-DITA Transform.....59
Word-to-DITA XSLT Transformation Parameters.....60
Word-to-DITA Ant Parameters.....61
The DITA-to-InDesign Transformation Framework.....62
Overview of the InDesign and InCopy Products.....64
Generating InDesign from DITA Using the Toolkit Plugin.....64

Configuring the DITA-to-InDesign Transformation....65
Preparing InDesign Templates for DITA-to-InDesign Use....68
This is an Appendix....72
Colophon....73

About This Book

This publication provides general guidance on how to use the various DITA for Publishers (D4P) materials, in particular, how to install and use the D4P Open Toolkit plugins and how to configure different XML editors to enable editing of D4P documents.

The DITA For Publishers markup is documented separately in the *DITA For Publishers Markup Guide and Reference*.

All DITA for Publishers materials are available through the DITA for Publishers project on SourceForge, *http://dita4publishers.sourceforge.net*.

Release Notes

Details about changes from release to release

The DITA for Publishers materials consists of the following major components. Each component has its own version number. The package of materials as a whole also has a version number, which is incremented for each release. For example, the vocabulary modules may be updated without the need to update any of the Toolkit plugins. In that case, the package and vocabulary version numbers are incremented but the Toolkit version numbers are not. All of the D4P Toolkit plugins report their individual version numbers when then run.

The components are:

- The DITA for Publishers vocabulary modules, packaged as the Open Toolkit plugin net.sourceforge.dita4publishers.doctypes.
- The Open Toolkit plugins for EPUB, Kindle, and HTML generation (the "HTML2" plugin).
- The Word-to-DITA transformation framework and Toolkit plugin
- The DITA-to-InDesign transformation framework and Toolkit plugin

The package also contains renditions of the *DITA for Publishers User Guide*.

The source materials for the DITA for Publishers project are available via anonymous Subversion access (including Web-based Subversion access) from *http://sourceforge.net/projects/dita4publishers/develop*.

Release Notes: Version 0.9.18

Version 0.9.18 released 8 Jan 2012

All transformation types (HTML2, EPUB, etc.)

- Removed duplicate XSLT imports
- Created common.html plugin to provide single import point for all extensions to the HTML transformation type
- Fixed handling of `@scale` attribute for images.

Word to DITA

- Fixed the following logged bugs:

3435135: Misnumbered topics	Added check for non-topic paragraphs between root-map-generating paragraph and first map- or topic-generating paragraph. This avoids the problem where content paragraphs immediately after map root cause misnumbering of generated topic filenames.
3435133: Use DOCX filename for name of result file	Updated the plugin Ant script to use the DOCX filename as the base name for the root topic or map by default.
3434693: An empty sequence is not allowed as the value of variable $styleMap	Fixed the code so the variable allows an empty sequence.
3435135: Simple word doc generates incorrect topic filename	Improved error reporting for non-topic-creating paragraphs before first topic-creating paragraph in Word doc.
3434694: Key styleMaps has not been defined	Fixed the XSLT.
3434693: An empty sequence is not allowed as the value of variable $s	Fixed the XSLT.

3436363: containerOutputclass causes transform to fail	Fixed in the XSLT.

- Implemented the following feature requests:

3309933: Add runtime parameter to set result language	Added new Ant parameter w2d.language that sets the value to use for the `@xml:lang` attribute. If not specified, value "en-US" is used.

- Other fixes:
  - Use configured topic extension for root topic output.
  - Use style name "Normal" for unmapped paragraphs. This lets you explicitly map unstyled paragraphs rather than depending on the built-in default mapping to `<p>`.
  - Updated built-in default style-to-tag map to include Heading 2, Heading 3, and Normal.
  - Refined the Ant script to make properties appropriately conditional and to avoid setting any unnecessary global properties.
  - Added start of an automated regression test suite in the project's test/ area in the source tree. These tests also serve as examples of how to use Ant to call the Toolkit transform.

EPUB and Kindle

- Fixed the following logged bugs:

3434065: Empty sequence not allowed for $graphicPath	Fixed the bug so `<image>` with no `@href` or `@keyref` is ignored when building the graphic map but is reported as a warning.
3433937: Include keywords as dc:subject in EPUB/Kindle books	Keywords within metadata/keywords become `<dc:subject>` elements in the EPUB and Kindle metadata.
3433935: Include authors in EPUB/ Kindle metadata	All authors in the publication metadata are included with appropriate roles.
3433934: Include ISBN and other bookid in EPUB/Kindle	ISBN numbers and other bookid content is be included in the EPUB and Kindle metadata appropriately.
3411763: Warnings received when an image has the @scale attribute set	Check for values with a leading "-" and suppress them and report them. If the value is positive, then use it. If it is a bare number, append "px" to the value.
3331319: @toc = no not respected for EPUB ToC	Topics with toc="no" should not be reflected in the EPUB ToC.
3317385: Chunked topics produce bad EPUB toc	Topics chucked to content should produce correct results in the ToC.
3312288: Problem with default parameter values	All D4P plugin Ant build files have been reworked to avoid setting any global parameters inappropriately.
3471010: Handle .jpeg files for EPUB/Kindle	EPUB and Kindle processing now correctly handles .jpeg extension in addition to .jpg.

- Other fixes/enhancements:
  - Factored out some common code between EPUB and Kindle transforms.

DITA to InDesign

- Enhancements and bug fixes to the Java INX support library. Can now generate multi-spread page sequences as configured in a separate configuration file. Corrects bugs with correlation of frames to pages.
- Added support for parsing and accessing conversion configurations defined using the new conversion_config topic type.
- Added XSLT support for generating CS 4/5 ICML InCopy articles from DITA content.

Vocabulary Modules

- Fixed the following logged bugs:

3371240: Inconsistencies in SYSTEM name in topic.dtd	Corrected "dp4" to "d4p" throughout. Corrected "d4pcommon" to "d4p_common".

- Corrected some configuration errors in doctype shells.
- Upgraded topic types to use DITA 1.2 coding conventions.
- Added new `<art-ph>` element. Specialize `<art>` from topic/p so it can go in `<fig>` directly.
- Added new topic type "conversion_configuration", used to configure Word2DITA and DITA2InDesign transforms, especially the DITA-to-InDesign Java processing.

Documentation

- Added missing documentation for word2dita Ant parameters.
- Added vocabulary topics from the website content into the User Guide

Release Notes: Version 0.9.16

Version 0.9.16 released 21 March 2011

Extensible transformation types (HTML2, EPUB, etc.)

- Corrected order of import extension point and base includes in top-level _template XSL files.
- Added sample no-op extension for the HTML2 transformation type (org.example.d4p.html2extensions).

Word to DITA

- Added new attribute, `@styleName` to `<style>` in style-to-tag-map documents. This allows you to define mappings in terms of Word style display names rather than style IDs. The `@styleId` attribute continues to be supported.

  This change addresses an issue seen with documents modified or created in Japanese versions of Word, where all the style IDs got changed from what they were originally.

  The documentation and samples have been updated to reflect the `@styleName` attribute.

Release Notes: Version 0.9.15

Version 0.9.15 released 22 February 2011

Common XSLT library

- Implemented decoding of escaped UTF-8 characters in URIs.

Word to DITA

- Fixed issue 3186860, Tables with no header row emit empty thead. Tables with no header rows and tables with only header rows should now produce valid DITA topics.

Vocabulary

- Corrected design problem with MathML integration. In order for the `<eqn-block>` and `<eqn-inline>` elements to be normal DITA elements, there must be another level of markup between those elements and the MathML `<math>` elements. I added a new container, specialized from `<foreign>`, named `<d4pMathML>`, that then contains the `<math>` element. This is the in d4pFormatting domain.

- Added `<object>` and `<foreign>` to content of `<art>`. This allows `<art>` to used to bind metadata to any kind of media object or display.

Release Notes: Version 0.9.14

Version 0.9.14 released 7 February 2011

HTML2 Transformation Types

- Corrected bug in generation of index.html file to use `name="contentwin"` in addition to `id="contentwin"`. This fixes an issue with handling of links from the dynamic ToC to the content topics displayed within the iframe.

Release Notes: Version 0.9.13

Version 0.9.13 released 5 February 2011

EPUB and Kindle Transformation Types

- Corrected bug that prevented the coverGraphicUri XSLT parameter from being set correctly by Ant.
- Corrected case of the idURIStub XSLT parameter to match the Ant script and the documentation (changed "IdURIStub" to "idURIStub".

Kindle Transformation Type

- Removed duplicate definition of global paramter idURIStub.

Release Notes: Version 0.9.12

Version 0.9.12 released 12 January 2011

HTML2, EPUB, and Kindle Transformation Types

- Fixed bug with footnote callouts getting suppressed in topic content.

Release Notes: Version 0.9.11

Version 0.9.11 released 9 January 2011

General

- Fixed bug where plugins unconditionally turned off graphic copying for all transformation types.

DITA to InDesign

- Added new DITA-to-InDesign transformation type.

  

  Note: The transformation type works but is not fully realized or completely documented.

- Added start of documentation for DITA-to-InDesign.

Vocabulary

- Added new experimental "variables" (d4p_variables) domain that is an experiment in using map and topic metadata to define "variables" that can be defined within a specific map or topicref context.

Release Notes: Version 0.9.10

Version 0.9.10 released 5 Novemeber 2010

EPUB, Kindle, and HTML2 Transformation Types

- Rewrite pointers to CSS style sheets in generated HTML files.

Word-to-DITA

- Added new parameters for suppressing tab and br elements from DITA XML.
- Added parameter to include the back pointers to the original Word document.

Vocabulary

- Corrected all doctype Toolkit plugin descriptors and top-level catalog.xml files to use dita.specialization.catalog.relative rather than dita.specialization.catalog.

Documentation

- Added more how-to and troubleshooting information for Word-to-DITA transform.
- Recorded video tutorial for Word-to-DITA style-to-tag mapping development. See *Word-to-DITA Style-to-Tag Mapping Video Tutorial* on page 57.

Release Notes: Version 0.9.9

Version 0.9.9 released 31 October 2010

EPUB, Kindle, and HTML2 Transformation Types

- Added support for different file organization strategies, with built-in strategies "single-dir" and "as-authored". Allows customization of organization and naming of output files through new mode "get-topic-result-url".
- Fixed bug with incorrect graphic reference URLs when topics are output into in directory under root directory or graphics not in directory named "images"
- Fixed bug with literal " " in empty table cells
- Handle paragraphs within span for table descriptions
- Eliminated Toolkit graphic copying during preprocessing

EPUB and Kindle Transformation Types

- Added new parameter epub.exclude.auto.rellinks that turns off auto-generated related links.

HTML2 Transformation Type

- Allow user to specify name of root output file (defaults to "index.html" as for HTML1 transform type)
- Use first navigation topic reference as initial file for root file and frameset
- Refined default CSS for root page

Vocabulary

- Added `<body-pullquote>` and `<section-pullquote>` to publication content domain. These elements specialize `<bodydiv>` and `<sectiondiv>` respectively.

Introduction to DITA For Publishers

The DITA for Publishers project applies the DITA standard and DITA technology to the specific requirements of Publishers as distinct from the requirements of technical documentation. The general goal of the project is to make creating and using DITA-based solutions for Publishing-specific business challenges as quick and easy as possible by providing a solid base from which you can start immediately.

DITA For Publishers is an open-source project, currently hosted on SourceForge. It is intended to be a community effort. The initial work on DITA for Publishers has been sponsored largely by Really Strategies, Inc.

By "Publishers" we mean enterprises whose primary business is producing authored material intended for reading by humans, e.g., books and magazines, usually where print is (or has been) the primary delivery medium. This category includes of course publishers of fiction and non-fiction trade books, magazines, journals, textbooks, nature and travel guides, and so on. It also includes groups within other enterprises who publish materials that are not product manuals or other very specialized information.

While DITA is often associated exclusively with technical documentation and highly modular information, DITA is a completely general standard and technology and can be applied to documents of any sort. DITA absolutely does not require modular writing or breaking all your content into small files or any other particular way of doing things. It does, of course, support those ways of doing things quite well, but it also supports other ways of doing things just as well.

As a technology, DITA offers a number of compelling advantages over other XML standards and approaches. In particular, it enables blind interchange of content while also allowing customized markup. This aspect of DITA is of vital importance to Publishers where the ability to interchange content with the lowest cost to all parties is of paramount importance. As a Publisher you want to be able to license your content to others and licence other's content. You want to be able to reuse components of publications in new packages as quickly and easily as possible. You want your content to have the highest value for the lowest cost.

DITA enables all of this. The premise of the DITA for Publishers project is that DITA, because of its unique design and architectural features, provides the lowest possible cost of startup and ownership and provides the highest possible value for interchange.

But for Publishers (or any enterprise) to be able to take advantage of this value there must be something to start with that works out of the box and that makes it practical to go forward. That is the goal of DITA for Publishers—to make getting started with DITA in a publishing context as quick and easy as it can possibly be.

DITA for Publishers does this by providing the following materials:

- A set of DITA vocabulary modules ("specializations") optimized for representing typical Publishing documents, namely books and magazines. The markup provided by these modules supports the business realities of Publishing, such as the fact that publications can be quite varied in their structure, that sometimes you have to capture arbitrary formatting, and on on.
- A set of plugins for the DITA Open Toolkit that support these vocabulary modules.
- An EPUB generation plugin for the DITA Open Toolkit, making it possible to generate publication-ready EPUBs from DITA-based content.
- A general-purpose Word-to-DITA transformation framework for converting manuscripts in Word into DITA, in order to support Word-primary editorial processes.
- A general-purpose DITA-to-InDesign transformation framework for generating InDesign articles and documents from DITA-based content.
- General knowledge and guidance on how to apply the DITA technology and tools to typical Publishing business processes.

The DITA For Publishers Open Toolkit Plugins

The DITA For Publishers (D4P) project provides a number of plugins for the DITA Open Toolkit. These plugins include:

- A plugin with all the D4P vocabulary modules and sample document type shells (net.sourceforge.dita4publishers.doctypes) or simply "the doctypes plugin".
- An EPUB generation plugin that allows generation of EPUBs from any DITA map (net.sourceforge.dita4publishers.epub)
- Plugins that provide D4P-specific extensions to the other standard Toolkit plugins (HTML, PDF, etc.). These plugins are named by the map, topic, or domain type they support plus the transformation type they extend, e.g. "net.sourceforge.dita4publishers.formatting-d.fo".

The D4P plugins are designed to themselves be extended using normal Toolkit extension mechanisms.

The D4P doctypes plugin integrates the D4P vocabulary modules with the Open Toolkit and thereby makes them available to any tool that does or can use the Toolkit's entity resolution catalog. For example, if you deploy the D4P doctypes plugin to the Toolkit integrated with OxygenXML, you can immediately start editing D4P maps and topics.

The doctypes plugin is the only plugin that is required in order to work with DITA for Publishers content. The other plugins may be used or not as you choose.

Installing the Toolkit Plugins

How to install the DITA for Publishers Toolkit plugins to the Open Toolkit.

All the plugins are distributed as Zip files. Installation is basically unzipping the Zips into the Toolkit's `plugins` directory and then running the `integrator.xml` Ant task.

1. Unzip the plugin's Zip file into your Toolkit's `plugins` directory.

   You should get one or more directories under `plugins` named like `net.sourceforge.dita4publishers.doctypes`.
2. Run the `integrator.xml` script from the Toolkit's root directory.

   The trick here is knowing how to access the "ant" command. If you open a command window, type "ant" and get something like

   ```
   Buildfile: build.xml does not exist!
   Build failed
   ```

   Then you are good to go.

   Otherwise, you have to either find where Ant is on your system or download Ant from *ant.apache.org*.

   If you are using the full install of the DITA Open Toolkit then ant is in `tools/ant/bin` below the Toolkit's root directory.

   If you have the OxygenXML editor installed then ant is in `tools/ant/bin` below Oxygen's installation directory (e.g., `c:\Program Files\Oxygen XML\tools\ant\bin` on Windows).

   Assuming you have ant in your Toolkit, you can run it like so:

   1. Open a command window and navigate to the root of your Toolkit (for this example, installed in `c:\programs\DITA-OT`:

      ```
      c:\> cd c:\programs\DITA-OT
      c:\Programs\DITA-OT>_
      ```

2. Run this command

```
c:\Programs\DITA-OT> tools\ant\bin\ant -f integrator.xml
```

You should see output something like this:

```
Buildfile: integrator.xml

integrate:
[integrate] Using XERCES.

BUILD SUCCESSFUL
Total time: 0 seconds
```

You may see messages about missing dependencies but as long as you get BUILD SUCCESSFUL then all the plugins that could be loaded have been.

The plugins should be ready to use.

Setting Up The kindlegen Command-Line Utility

The Amazon-supplied kindlegen utility is required in order to produce final-form Kindle books.

While an EPUB is just a Zip file, a Kindle book is in a proprietary format that can only be produced from the command line using the Amazon-provided kindlegen utility. The kindlegen utility is a free utility but is proprietary so it cannot be packaged with either the Open Toolkit or the DITA For Publishers plugins.

To get kindlegen, google "kindlegen" and the Amazon download page will probably be the top hit.

Download the package and follow the installation instructions (which should be to simply unpack it somewhere convenient).

You can either put the kindlegen executable's directory in your PATH environment variable or you can specify the full path to the executable using the kindlegen.executable Ant parameter (see *kindlegen.executable Ant parameter* on page 21).

Generating EPUBs and Kindle Books from DITA Content

The D4P EPUB and Kindle book generation Open Toolkit plugins enable generation of delivery-ready EPUBs and Kindle books from any conforming DITA content.

The DITA for Publishers EPUB generation plugin (net.sourceforge.dita4publishers.epub) generates EPUBs that conform to the IDPF EPUB standard. The plugin works with any conforming DITA content. The resulting EPUBs may be used with any EPUB reader software or hardware. The EPUBs have been tested on a number of devices and readers, including iBooks and Adobe Digital Editions.

The DITA for Publishers Kindle book generation plugin (net.sourceforge.dita4publishers.kindle) generates Kindle books (Mobipocket format) that can pass the kindlegen checks on Kindle book content and organization. The plugin works with any conforming DITA content. The resulting Kindle books may be used any Kindle reader or published through Amazon. The Kindle books have been tested with several different Kindle emulators and Mobipocket readers, as well as by publishing a sample book through Amazon's Kindle publishing process.

See *Installing the Toolkit Plugins* on page 12 for instructions on how to install the EPUB and Kindle plugins to the DITA Open Toolkit.

The plugins do not require any specific markup in order to get a usable EPUB or Kindle book. However, the plugin does take advantage of specific markup in order to generate EPUB- and Kindle-specific features such as cover graphics, book identifiers, and so on.

The EPUB plugin defines the transformation type "epub" (Ant target "dita2epub"). It is based on the standard HTML transformation type and therefore takes all the same parameters as the HTML transformation type. All EPUB-specific parameters are optional.

The Kindle plugin defines the transformation type "kindle" (Ant target "dita2kindle"). It is based on the standard HTML transformation type and therefore takes all the same parameters as the HTML transformation type. All Kindle-specific parameters are optional. The Kindle generation process attempts to use the Amazon-provided "kindlegen" command-line tool to construct a publishable Kindle book. You must acquire and install kindlegen separately. The kindlegen tool is proprietary Amazon software and therefore is not licensed in a way that is consistent with the Open Toolkit's open source license. The kindlegen tool can be downloaded from Amazon. Google "kindlegen" to find the exact download location.

You can use the EPUB transformation type by simply specifying a transtype value of "epub" rather than "html" and use the same parameters you would use for HTML generation.

You can use the Kindle transformation type by simply specifying a transtype value of "kindle" rather than "html" and use the same parameters you would use for HTML generation.

Generating EPUBs From OxygenXML

You can configure a DITA transformation scenario in OxygenXML to generate EPUBs.

You must have the EPUB plugin deployed to the Open Toolkit used by OxygenXML, normally in the `frameworks/dita/DITA-OT` directory under Oxygen's installation directory. You can also use a different Toolkit by changing the value of the dita.dir Ant parameter.

These instructions reflect Oxygen version 11.2 and 12.0.

To set up an EPUB transformation scenario in OxygenXML, do the following.

1. From either the main editing window or the DITA Maps Manager, open the "Configure Transformation Scenario" dialog (the little wrench and red arrow button in the toolbar).
2. Select "DITA OT Transformation" from the **Scenario Type** pulldown.
3. Select the "DITA Map XHTML" transform and press the **Duplicate** button.

   The "Edit Scenario" dialog should open.
4. Change the scenario name to something like "DITA Map EPUB"
5. Select the **Parameters** tab (it should be selected by default), add a new parameter named "transtype" and set its value to "epub".
6. Select the **OK** button to save the scenario.

   You should see your new scenario in the list of available DITA OT transformation scenarios.

The scenario should be ready to use. You can test it by applying it to any DITA map. The EPUB file is generated in whatever you've configured as the output directory using the normal output location configuration (as specified in the Output tab of the transformation scenario configuration dialog.

If you select the "Open in browser" option in the transformation scenario's output options and you have an application associated with .epub files, such as Adobe Digital Editions, the newly-generated EPUB should open in that application. Note that in OxygenXML 12 you have the option of associating EPUB files with OxygenXML, which can open them in its Archive Browser. If you have that association, then doing "Open in browser" will open the EPUB in OxygenXML, not in an EPUB viewer.

You can edit the scenario to add additional EPUB parameters as necessary. Simply open the scenario in the Scenario Editor, select the "add parameter" button, and create the parameter. The EPUB-specific parameters are documented in *EPUB and Kindle Transformation Parameters* on page 19.

Generating Kindle Books From OxygenXML

You can configure a DITA transformation scenario in OxygenXML to generate Kindle books.

You must have the Kindle plugin deployed to the Open Toolkit used by OxygenXML, normally in the `frameworks/dita/DITA-OT` directory under Oxygen's installation directory. You can also use a different Toolkit by changing the value of the dita.dir Ant parameter.

For the .mobi file generation to work you must have the Amazon-provided kindlegen tool installed on your system. It must either be in the PATH environment variable or you must specify the full path to the executable as the value of the kindlegen.executable Ant parameter.

These instructions reflect Oxygen versions 11.2 and 12.0.

To set up a Kindle book transformation scenario in OxygenXML, do the following.

1. From either the main editing window or the DITA Maps Manager, open the "Configure Transformation Scenario" dialog (the little wrench and red arrow button in the toolbar).
2. Select "DITA OT Transformation" from the **Scenario Type** pulldown.
3. Select the "DITA Map XHTML" transform and press the **Duplicate** button.

   The "Edit Scenario" dialog should open.
4. Change the scenario name to something like "DITA Map Kindle"
5. Select the **Parameters** tab (it should be selected by default), add a new parameter named "transtype" and set its value to "kindle".
6. If the kindlegen executable is not included in your system's PATH environment variable, add a new parameter named "kindlegen.executable" and set its value to the full path to the executable, e.g. "/Users/ekimber/apps/KindleGen_Mac_i386_v1.1/kindlegen".
7. Select the **OK** button to save the scenario.

   You should see your new scenario in the list of available DITA OT transformation scenarios.

The scenario should be ready to use. You can test it by applying it to any DITA map. The Kindle file is generated in whatever you've configured as the output directory using the normal output location configuration (as specified in the Output tab of the transformation scenario configuration dialog.

If you select the "Open in browser" option in the transformation scenario's Output panel and you have an application associated with .mobi files, such as the Kindle Previewer or the Kindle book reader application, then the newly-generated book should open in that application.

The Kindle generation process and resulting data is almost identical to the EPUB output, so all the EPUB parameters are also relevant for Kindle output.

You can edit the scenario to add additional EPUB or Kindle parameters as necessary. Simply open the scenario in the Scenario Editor, select the "add parameter" button, and create the parameter. The EPUB-specific parameters are documented in *EPUB and Kindle Transformation Parameters* on page 19.

Using Custom CSS Style Sheets

You can include custom CSS styles sheets in your EPUB just as you can for HTML output generally.

Note: At the moment, this mechanism only supports a single CSS file. A future enhancement will be to allow the copying of all the CSS files in a specified directory.

The EPUB transformation type uses the same Ant parameters for CSS styles sheets as for the normal HTML transformation type. The difference is that you *must* copy the CSS files to the output since they need to be part of the EPUB package. Thus the args.copycss parameter is implicitly set to "yes" if you specify the args.css parameter.

Likewise, the parameter args.csspath is ignored because the EPUB processor controls how the CSS files are referenced in the generated HTML files.

You may specify the directory within the EPUB package to contain the CSS files using the parameter args.css.output.dir. By default it is the same as the topic output directory (which by default is "topics").

The args.css parameter can be an absolute path to the CSS file or it can be a path relative to the value of the args.cssroot parameter.

For example, if you have a custom CSS file `mystyle.css` in the directory `c:\projects\mydoc\css` and you want to use that CSS file in your EPUB you would specify the Ant parameter args.css with the value "c:\projects\mydoc\css\mystyle.css".

If you have a common CSS source directory and you use different CSS files for different publications, you can specify both args.cssroot and args.css like so:

- args.cssroot as "c:\projects\mydoc\css"
- args.css as "mystyle.css"

If you want your CSS somewhere other than the topics directory, specify the css.output.dir parameter to the desired directory name, e.g. "css".

The built-in CSS style sheets provided with the DITA Open Toolkit are always included in the EPUB in the CSS output directory and are always referenced in the HTML after any user-specified CSS style sheet.

Creating Cover Graphics and Covers

Some readers and book libraries, such as iBooks, have special conventions for cover graphics.

Some EPUB readers and library systems have special conventions for covers and cover graphics that are used to provide a graphic or icon for the book. In addition, you may want to generate or provide a cover page that serves as the first page in play order for the EPUB.

For example, with iBooks, if you put the metadata entry `<meta name="cover" content="coverimage"/>` in your EPUB's OPF file (the main manifest), where "coverimage" is the manifest ID of an included graphic, then iBooks will use that graphic in the iBooks bookshelf in iTunes and in iBooks, rather than the default cover graphic. Other readers and EPUB libraries have similar features.

The EPUB transform provides several ways to specify the cover graphic:

- You can specify the graphic as a parameter to the generation process (*cover.graphic.file Ant parameter* on page 19)
- You can include a `<data>` element with a @name value of "covergraphic" in the root map's `<topicmeta>` element, where the value or content of the `<data>` element is the absolute or relative path (from the map) of the graphic to use.
- You can use the DITA for Publishers `<epub-cover-graphic>` element within the `<covers>` section of a publication map to point to the cover graphic.
- You can implement a Toolkit plugin that extends the base EPUB generation transform and override the template for `<map>` in the mode "get-cover-graphic-uri". For example, you may have an existing convention for representing cover graphics in your publications.

Note: The cover graphic used for the EPUB cover must (and should) be different from any other graphic representation of the cover. For example, if you reference the same graphic as the EPUB cover graphic and as an image from a "cover" topic, the EPUB OPF metadata entry will probably not be correct.

As a general practice, the EPUB cover graphic should be a fairly low-resolution image sized according to the guidelines for the target EPUB reader or library.

TBD: Tips about graphics for EPUBs in different readers. If anyone knows anything useful, please let me know—Eliot.

Creating Arbitrary EPUB OPF Metadata

The OPF package manifest within an EPUB document can contain metadata entries. The entries required for EPUB validation are created automatically by the EPUB generation process. However, you may want or need to add additional metadata, for example to support the conventions of a particular EPUB reader.

You can create arbitrary OPF metadata entries by creating a `<data>` element in your map's metadata with the name "opf-metadata". Use child `<data>` elements to define specific metadata items, where the `@name` value is is used as the `@name` attribute on the OPF `<meta>` and the `@@value` or content of the `<data>` element is used as the `@content` attribute on the OPF `<meta>` element.

For example, this markup in a DITA map:

```
<map>
 <topicmeta>
```

```
  <data name="opf-metadata">
    <data name="item-one" value="value one"/>
    <data name="item-two" value="value two"/>
  </data>
 </topicmeta>
</map>
```

Would result in OPF metadata like this:

```
   ...
   <metadata>
      <dc:title>DITA For Publishers User Guide</dc:title>
      <dc:language id="language">en-US</dc:language>
      <dc:identifier id="bookid">http://example.org/epubs/</dc:identifier>
      <dc:creator opf:role="aut">W. Eliot Kimber</dc:creator>
      <meta name="item-one" content="value one"/>
      <meta name="item-two" content="value two"/>
   </metadata>
   ...
```

Implementing Overrides for Common Processing

You can use the same args.xsl parameter as for the XHTML transform to specify a custom top-level transform that includes overrides to the base processing.

The Toolkit's XHTML transform provides a parameter, args.xml, by which you can specify the top-level XSLT transform to be used by the transformation type. You normally use this facility to apply overrides to the base processing provided by the transformation type.

Note that there is an important distinction between *extending* a transformation type and *overriding* a transformation type. Extensions are provided through Toolkit plugins that are integrated with the base transformation type and apply to all uses of that transformation type. For example, the DITA for Publishers formatting-d.html and formatting-d.fo Toolkit plugins extend the base XHTML and PDF transforms with support for the DITA for Publishers formatting domain elements. These extensions are used unconditionally by all invocations of the XHTML and PDF transformation types.

When you *override* a transformation type you are changing the base processing for a specific invocation of the transformation type. Overrides are not integrated with the base transformation type. Rather, they are implemented as new top-level XSLT transforms that import the base transformation (e.g., dita2xhtml.xsl) and include or import custom XSLT code that extends or overrides the base processing.

You normally use overrides to achieve output results that are specific to a particular document, document type, or delivery target. For example, you might want to output specific topic-level metadata for a specific kind of publication or even for a specific single publication. When dealing with Publishing use cases, where every publication might have different requirements it is likely that you may need to implement publication-specific overrides to achieve some specific effect.

Note also that the general override strategy for HTML-based outputs like EPUB and Kindle is different from the override strategy for PDF output using the Toolkit's built-in PDF transformation type. The PDF transformation type has its own customization framework that provides a different way to apply overrides to the transformation (it also allows extension using integrated plugins).

You can override the EPUB and Kindle processing by creating a new top-level XSLT transformation that includes the base map2epub.xsl or map2kindle.xsl transforms provided by the respective Toolkit plugins and that also includes your custom XSLT transform that does whatever you need. Because the EPUB and Kindle transforms are both based on the XHTML transform you can normally use a single override file with the XHTML, EPUB, and Kindle transformation types.

The top-level transformation can be packaged as a separate Open Toolkit plugin that depends on, but does not integrate into, the base transformation type it extends. This is the best thing to do if you must distribute the extension to different systems. If the override will only be used on a single machine you can put the top-level transformation

anywhere. However, in this case the top-level transformation will necessarily have system-specific URIs for the Toolkit-provided base transformations, which means it is not portable.

A typical "local" top-level transformation file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  version="2.0">

  <xsl:import href="/Applications/oxygen/frameworks/dita//DITA-OT/plugins/
net.sourceforge.dita4publishers.epub/xsl/map2epub.xsl"/>
  <xsl:include href="my-html-overrides.xsl"/>

</xsl:stylesheet>
```

This example is a "local" override that uses an absolute URI to import the base transformation, in this case the EPUB `map2epub.xsl` transform. It also includes the local overrides, here in the file `my-html-overrides.xsl`. The corresponding transformation for the Kindle transformation type would include the Kindle-specific top-level transform and the same overrides (assuming that you want the EPUB and Kindle outputs to reflect the same overrides, which would normally be the case).

The overrides themselves can be anything you need and can override either the base XHTML processing from the dita2xhtml transformation type or can override the EPUB- and Kindle-specific processing from the EPUB or Kindle plugins. For most overrides you would be overriding the base XHTML processing.

To use the override transformation you simply specify the path and filename of the transformation as the value of the args.xml Ant parameter for the XHTML, EPUB, or Kindle transformation types.

To package an override as a Toolkit plugin you create a `plugin.xml` file that defines the plugin and use a relative URI to include the base transformation type, relative to the plugin's location as deployed in a Toolkit. You put the `plugin.xml` and override transform into their own uniquely-named directory within the Toolkit's `plugins/` directory.

The `plugin.xml` file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin id="my.unique.plugin.name">

  <require plugin="net.sourceforge.dita4publishers.epub"/>

</plugin>
```

Where the bold text should reflect a unique name for this particular plugin.

This plugin descriptor simply establishes a dependency on the Toolkit transformation type it overrides, e.g., the EPUB plugin.

Assuming that you are deploying the plugin to the `plugins/` directory and that your override template is at the root of its own plugin's directory, then the top-level transform should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  version="2.0">

  <xsl:import href="../net.sourceforge.dita4publishers.epub/xsl/
map2epub.xsl"/>
  <xsl:include href="my-html-overrides.xsl"/>
```

```
</xsl:stylesheet>
```

With this plugin you can then specify the transformation within your plugin as the value of the args.xml Ant parameter.

EPUB and Kindle Transformation Parameters

EPUB-specific parameters for the EPUB transformation type

The EPUB transformation type allows a number of parameters that configure the details of the generated EPUB. Available options include:

- Include a back-of-the-book index
- The names of the directories used to hold topics, graphics, and CSS style sheets.
- The topic and title element @class values to use for topics generated from topicheads.
- The maximum depth of the generated table of contents.
- The graphic to use as the EPUB cover graphic (<meta name="cover" content="coverimage" /> as used by iBooks) if the cover image is not indicated by markup in the map.
- The EPUB publication ID
- Do not treat external-scope xrefs as links in the HTML (avoids EPUB validation issues with unavailable resources).
- Include user-specified CSS files in the generated EPUB package.

The parameters documented in this section are Ant parameters recognized by the `build_transtype_epub.xml` Ant script. Each of these Ant parameters corresponds to an XSLT parameter used by the underlying XSLT transforms. The XSLT parameters are defined in the `map2epubImpl.xsl` transform in the EPUB plugin's `xsl/` directory.

args.css.output.dir Ant parameter

Specifies the directory to put CSS files into for EPUB and Kindle transforms.

Value

An absolute platform-specific filename.

Default

The configured topic output directory.

cover.graphic.file Ant parameter

Specifies the absolute or relative path of the graphic to use as the cover graphic for the EPUB.

Value

An absolute, platform-specific filename or a path relative to the root map to a JPG, PNG, or GIF to use as the cover graphic.

Default

"" (no cover graphic or cover graphic determined by markup in the input document).

XSLT Parameter

coverGraphicUri (via Ant's conversion of filenames into URIs).

Notes

See *Creating Cover Graphics and Covers* on page 16 for more information about EPUB-specific cover graphics and alternative ways of specifying the graphic to use.

epub.exclude.auto.rellinks Ant parameter

Turns on exclusion of auto-generated related links (as for normal HTML output)

Value

Any value (e.g. "true") sets the property to true.

Default

Not set (related links are generated)

XSLT Parameter

None.

Notes

Controls base Toolkit processing by turning off the maplink-check and maplink targets, which has the effect of eliminating auto-generated related links. Note that explicitly-authored related links are not affected. This property cannot be set to true by default because it is a global property that affects all transformation types. You must set this property explicitly in order to suppress related links in the output.

epub.pubid.uri.stub Ant parameter

Specifies the URI prefix stub to use in constructing the full EPUB ID value in the EPUB metadata.

Value

Any valid URI string. If this is a relative URI it should end with a "/" character.

Default

"http://my-URI-stub/" (this is a placeholder that will be flagged by the EPUB transform).

XSLT Parameter

idURIStub

Notes

This parameter should normally reflect the organization that owns or publishes the EPUB, e.g., "http://mydomain.com/EPUB" or something similar.

epub.temp.dir Ant parameter

Specifies the directory to use for EPUB-specific processing separate from the main Toolkit's temp directory.

Value

An absolute platform-specific filename.

Default

directory `epub_temp` within the defined Toolkit temp directory.

Notes

This directory holds all the files that are then Zipped up to create the final EPUB file.

images.output.dir Ant parameter

Specifies the name of the directory within the EPUB to hold all images.

Value

Any valid directory name

Default

"images"

XSLT Parameter

imagesOutputDir

Notes

The EPUB process automatically renames graphics to ensure that there are no name conflicts when the graphics are copied to into this directory from their source locations.

kindlegen.executable Ant parameter

Specifies the name, and if necessary, full path of the kindlegen command-line executable

Value

The absolute path to the "kindlegen" executable (available from Amazon).

If the kindlegen executable is in your PATH environment variable and it is named "kindlegen" then you should not need to specify this parameter.

If the kindlegen executable is in your path but is named something other than "kindlegen" then you can specify just the name of the executable, not the entire path.

Default

"kindlegen"

Notes

See *Setting Up The kindlegen Command-Line Utility* on page 13.

title.only.topic.class.spec Ant parameter

Specifies the @class attribute value to use for title-only topics generated from topicheads.

Value

Any valid @class value for a topic type (e.g., `"- topic/topic mytopic/mytopic "`).

Default

`"- topic/topic "`

XSLT Parameter

titleOnlyTopicClassSpec

Notes

The EPUB standard requires that all navigation ToC entries point to a content object. The EPUB transform handles this by generating title-only topics for each topichead in the map. These generated topics are then processed by the normal HTML generation process.

If you have a topic specialization that produces custom HTML results that you want to use for these generated topics, you can use this parameter to generate topics of that type. Note that this parameter applies to all generated title-only topics—there is currently no mechanism to get different topic types for different map contexts.

title.only.topic.title.class.spec Ant parameter

Specifies the @class attribute value to use for titles for title-only topics generated from topicheads.

Value

Any valid @class value for a topic title (e.g., `"- topic/title mytopic/mytitle "`).

Default

```
"- topic/title "
```

XSLT Parameter

titleOnlyTopicTitleClassSpec

Notes

The EPUB standard requires that all navigation ToC entries point to a content object. The EPUB transform handles this by generating title-only topics for each topichead in the map. These generated topics are then processed by the normal HTML generation process.

If you have a topic title specialization that produces custom HTML results that you want to use for these generated topics, you can use this parameter to generate titles of that type. Note that this parameter applies to all generated title-only topics—there is currently no mechanism to get different title types for different map contexts.

topics.output.dir Ant parameter

Specifies the name of the directory within the EPUB to hold all the topics.

Value

Any valid directory name.

Default

"topics"

XSLT Parameter

topicsOutputDir

Notes

The HTML files generated for topics are given unique names within the output directory so that there are no name conflicts.

HTML2 Plugin

Generating HTML using map-drive processing

The HTMLT2 Toolkit extends the original Toolkit's HTML generation process by adding a map-driven process whereby the HTML generation for each topic is driven by processing of the root input map. This allows each topic to be processed with full knowledge of its use context in the map. This in turn enables a number of important functions:

- Numbering of topics and within-topic elements in terms of the topicref hierarchy in the map ("enumeration").
- Processing of topics in terms of semantics imposed by topicrefs (e.g., processing a generic topic as a "chapter" because it is referenced by a `<chapter>` topicref.
- Distinguishing separate uses of the same topic within a map in order to do automatic chunking and link rewriting.
- Generating HTML files in a structure different from the source file organization structure.
- Generation of literal HTML files for topic heads as though the topic heads were references to title-only topics.
- Generation of a back-of-the-book index and other publication-scope lists.

The HTML2 transform uses the base HTML transformation type for general topic HTMl generation. It accepts all the parameters for the base HTML transform in addition to the HTML2-specific parameters documented in this section.

Understanding the HTML2 Transformations

Guidance on how the HTML2 XSLT transformations are organized and how the processing works.

The HTML2 transformations are "map driven", meaning that the initial input document is a resolved DITA map and all processing of the map and its referenced topics is driven directly by the map structure. This processing takes advantage of XSLT2 features that make it easier to do multi-stage processing within a single XSLT transformation and that make it possible to produce multiple result documents. The transform also takes advantage of XSLT2's support for locally-defined functions to make it possible to do complex filename processing, which enables more sophisticated and flexible output processing than the original HTML transformation type can provide (which I will refer to as "HTML1" in this document).

The HTML2 transformation type does use the general topic-to-HTML processing provided by the HTML1 transformation type, meaning that any plugin-provided extensions to the HTML transformation type will be used by the HTML2 transformation type.

The HTML2 transformation type replaces all the topic file and graphic file list generation required by the HTML1 topic type.

Understanding and extending the HTML2 transformation type requires use of and understanding of XSLT2 features.

The HTML2 transforms have been designed for maximum ease of extension and customization. They make heavy use of XSLT modes.

For the purposes of this discussion, the map and its local-scope dependencies are referred to collectively as "the publication". Because the transform processes everything in one go, it is sensible to think of the compound document defined by map as a single entity for processing purposes. That entity is "the publication".

Overall HTML Generation Process Flow

Describes the overall processing flow for the HTML2 transformation

The root transformation file is `map2html2Impl.xsl`. This transform defines all the runtime parameters, sets up global variables and handles the initial map processing. It contains a simple passthrough template for "/" and the default-mode template for map.

The default-mode map template does the following tasks:

- Constructs a list of the graphics used within the publication
- Constructs an intermediate file that maps the graphic locations as authored to their output locations relative to the generated HTML files. This allows the graphics to have different filenames and locations in the output (for example, all graphics can be put into a single output directory regardless of how they are stored for authoring).
- Constructs an intermediate data set for the back-of-the-book index that can then be used to generate various renditions of the index.
- Processes the map in the mode "generate-root-pages" in order to generate the root page or pages, such as the "index.html" file that contains navigation structures, a frameset if appropriate, and so on. This mode in turn does

generation of static and dynamic tables of contents as selected by the runtime parameters. By default a dynamic ToC is generated and a static ToC is not.
- Processes the map in the mode "generate-content", which manages the generation of HTML from topics as driven by the map structure.
- Processes the map in the mode "generate-index", which manages the generation of a back-of-the-book index as a separate HTML page or pages.
- Processes the map in the mode "generate-graphic-copy-ant-script", which generates an Ant script that does the copying of graphics from their source locations to their output locations.

Most of this processing is fairly straightforward as it is simply processing of topicref-type elements in various contexts to produce different outputs. There is no particular non-obvious processing going on.

The non-obvious processing occurs for the processing of topics in the context of their references within the map. In order to enable use of base XSLT transforms for topics while still providing map-context-aware processing that can be overridden easily, the code uses a somewhat convoluted "double dispatch" method.

The XSLT file `map2html2Context.xsl` iterates over the list of unique topic document references, as defined by the utility function df:getUniqueTopicrefs(), which takes into account chunking behavior. For each unique topic document reference, processing is applied in the mode "generate-content".

The template for topic references (as opposed to topic heads or topic groups) resolves the topicref to a topic document. It then first processes the topic in the mode "href-fixup", which is implemented by the common module "topicHrefFixup.xsl in the "net.sourceforge.dita4publishers.common.xslt" Toolkit plugin. That mode is an identity transform that rewrites all `@href` attribute values to reflect the eventual output locations of all the output files, resulting in an in-memory temporary topic document that is then processed in the mode "generate-content", passing the topicref element and result URI as tunnel parameters to the apply-templates call.

The generate-content template for topics then applies templates in the mode "map-driven-content-processing" to generate the initial no-namespace HTML file generated by the base HTML1 transforms, captured into a temporary variable. The no-namespace HTML is then processed in the mode "no-namespace-html-post-process" to produce the result file, which by default simply copies the no-namespace HTML to the output. You can override this mode to further process the HTML, for example to generate XHTML or HTML5 (as is done by the EPUB and HTML5 transforms, respectively).

The "map-driven-content-processing" template for topics checks to see whether or not the topicref parameter has a value. If it does, it applies templates in the mode "topicref-driven-content" to the topicref passed in as a parameter, passing the *topic* as a parameter to apply templates. If the topicref parameter does not have a value, then it applies templates to the topic in the default mode, which has the effect of applying the normal HTML1 processing to topic in order to generate non-namespaced HTML.

The "topicref-driven-content" template for topicref then applies templates to the *topic* in the default mode, passing the *topicref* as a tunnel parameter. This has no effect on the (current) base HTML1 transforms but makes the topicref available to any extensions that choose to grab it.

The result URLs for files generated from topics (the HTML files) are determined by by the function htmlutil:getTopicResultUrl(). If the global file organization strategy is "single-dir" then the function calls the named template "get-topic-result-url-single-dir" otherwise it applies templates to the map in the mode "get-topic-result-url", which by default applies the "as-authored" strategy. You can implement your own template that matches on map/map in the "get-topic-result-url" mode or you can override the "get-topic-result-url-single-dir" template to implement your own output file organization strategy.

This rather convoluted processing chain is necessary in order to enable direct use of the HTML1 transforms with the option of map awareness. If the HTML1 transforms were reimplemented as XSLT2 transforms then at least some of this mode switching would not be necessary.

This profusion of XSTL modes also enables override and extension at several points and enables both pre-processing of the topics before they are sent to the HTML processing (by hooking the href-fixup mode) and post-processing of the initially-generated HTML (by overriding the no-namespace-html-post-process mode).

Extending and Customizing Enumeration Processing (Numbering)

How to customize and extend the default enumeration processing for topic titles and other numbered things.

TBD

HTML2 Transformation Parameters

Unique parameters for the HTML2 transformation type

The HTML2 transformation type allows a number of parameters that configure the details of the generated HTML. Available options include:

- Include a back-of-the-book index
- The names of the directories used to hold topics, graphics, and CSS style sheets.
- The maximum depth of the generated table of contents.

The parameters documented in this section are Ant parameters recognized by the `build_transtype_html2.xml` Ant script. Each of these Ant parameters corresponds to an XSLT parameter used by the underlying XSLT transforms. The XSLT parameters are defined in the `map2html2Impl.xsl` transform in the HTML2 plugin's `xsl/` directory.

html2.file.organization.strategy Ant parameter

Names the output file organization strategy to use for generated files.

Value

One of the pre-defined strategies "as-authored" or "single-dir" or a user-implemented name. If the strategy name is not recognized (that is, not handled by the base code or an extension) then the default strategy is applied. The default is dependent on the specific transformation type. For HTML2 the default is "as-authored". For EPUB it is "single-dir".

Default

Unset, in which case the transform-type-defined default strategy is used.

XSLT Parameter

fileOrganizationStrategy.

Notes

html2.generate.dynamic.toc Ant parameter

Specifies whether or not to generate a dynamic TOC in the output.

Value

"true" or "false"

Default

"true" (generate a dynamic table of contents).

XSLT Parameter

generateDynamicToc.

Notes

The dynamic ToC uses JavaScript to display a dynamic tree view of the ToC.

html2.generate.frameset Ant parameter

Specifies whether or not to generate an HTML frameset file for the map.

Value

"true" or "false"

Default

"false" (do not generate a frameset).

XSLT Parameter

generateFrameset.

Notes

The use of framesets is generally deprecated as HTML practice and the frameset elements are deprecated in the latest HTML standards.

html2.generate.static.toc Ant parameter

Specifies whether or not to generate a static TOC in the output.

Value

"true" or "false"

Default

"false" (do not generate a static table of contents).

XSLT Parameter

generateStaticToc.

Notes

The static ToC uses JavaScript to display a static tree view of the ToC.

Generating DITA from Documents (Word-to-DITA Transformation Framework)

The DITA for Publishers Word-to-DITA framework is packaged as an Open Toolkit plugin.

The Word-to-DITA transformation framework enables reliable generation of DITA maps and topics from styled Word documents. See *The Word-to-DITA Transformation Framework* on page 40 for details. It can be used as a standalone XSLT transform or can be run against Word documents using the Open Toolkit.

The DITA For Publishers Markup Vocabulary

The DITA for Publishers project provides a set of map and topic types along with a number of markup domains specifically designed to meet the needs of Publishers and publishing-type documents (books and magazines).

This part provides guidance and reference on how to use this markup.

The key features of the DITA For Publishers markup vocabulary are:

- The publication map domain, which provides elements designed to represent a wide variety of publication structures at an appropriate level of detail.
- A publication metadata domain, which provides the metadata elements needed for publications, including sophisticated licensing modeling (including non-copyright-based licenses) and support for ONIX and Dublin Core metadata.
- A set of basic topic types for the components of publications: `<part>`, `<chapter>`, `<article>`, `<subsection>`, and `<sidebar>`. With these topic types, along with the generic DITA `<topic>` topic type, you can easily capture the content of any publication.
- A set of markup domains ("mix in" elements) that provide elements for arbitrary formatting, verse, and constructs typically found in fiction and trade books, such as epigraphs, and "enumerations", which enable capturing numbering from legacy content so it can be recreated as needed or used to enable searching.

Because this is DITA you can of course combine the DITA For Publishers vocabulary modules with any other DITA vocabulary modules. The following standard DITA modules are usually quite useful, if not essential, for Publishing content:

- The glossary entry (`<glossentry>`) topic type, which models glossary entries and enables sophisticated linking from mentions of terms in content to glossaries. Can support automatic construction of glossaries and automatic sorting and grouping of glossaries.
- The Learning and Training assessment domain, which provides markup for test questions. This markup can be used to both drive visual rendering of questions (e.g, printed tests or quizes) and also drive interactive assessments (e.g., via SCORM-based learning management systems).
- The Learning and Training map and topic types, which model common instructional design practice and organization.

Understanding the DITA For Publishers Markup Vocabulary

This section provides general concepts and usuage guidance for the DITA for Publishers vocabulary.

The DITA for Publishers vocabulary addresses the following key requirements that most Publishers have for most publications:

- Publication structures are almost infinitely variable across titles.

  This means that DITA maps that represent publications must be very flexible and impose few, if any, abitrary constraints. The Publication Map (pubmap) domain provides this level of flexibility.
- The rhetorical role of a given topic included in a publication is usually a function of how it is used, not of the topic itself.

  That is, unlike technical documentation written with a clear separation of information types (concept, task, reference), most publishing content is quite generic. Thus a given topic may function as an article in the context of an issue of a periodical, a chapter in the context of one book, and an appendix or sidebar in the context of another book.

  This means that publication maps must be able to impose onto the topics they use the rhetorical role those topics play in the that publication. The publication map domain supports this by providing a wide variety of topicref specializations that represent common publication components: chapter, article, part, subsection, sidebar, preface,

acknowledgements, etc. It also provides a pattern and example for creating additional such specializations as needed to support the needs of new publications.

- The metadata for publications is both sophisticated and varied.

  Unlike technical documentation, the metadata for publications is often quite extensive and variable from publication type to publication type and from title to title and from publisher to publisher.

  The Publication Metadata (pubmeta) domain provides a rich set of metadata elements that reflect the specific needs of commercial publications. As a domain it can be replaced in map document types with other metadata models while still using the pubmap domain. It also provides a base for further extension and specialization. This maximizes the flexibility of the metadata model for publication maps.
- Publications often have arbitrary and unpredictable content and formatting requirements.

  In technical documentation one goal is usually consistency of content and formatting across titles. In Publishing the exact opposite is usually the goal, or at least the reality, namely allow and encourage creative difference in order to distinguish publications from each other.

  The formatting and enumeration domains provide markup for capturing arbitrary formatting and numbering. The base Publishing topic types are based directly on the base DITA `<topic>` topic type, which imposes few constraints on the body content of topics. These two features enable capturing and expressing essentially unbounded variation in content structure and formatting requirements, as appropriate.

  On the other hand, where consistency is a goal, for example in series publications like travel guides or textbooks, you can use all the normal features of DITA to define map and topic models that impose the appropriate rules and help to ensure consistency of structure and content (keeping in mind that there is no substitute for editorial oversight and author training).
- The content of publications is pretty simple, except when it isn't.

  The content of most books and periodical articles is very simple: paragraphs, lists, tables, and figures. For this content a generic DITA topic with no additional domain modules beyond the highlight domain is sufficient. But some publications have topics that are quite sophisticated in their content. For that content you may want more specialized markup.

  DITA for Publishers provides out of the box a set of completely generic topic types that map to the basic components of most publications: part, chapter, article, subsection, and sidebar. These topic types are specialized directly from `<topic>`, differing from `<topic>` only in the root tagname. But because this DITA you can also use other topic types or define new topic types that provide more-precise markup as needed. Using normal DITA facilities you can integrate any needed DITA for Publishers domain modules into those topic types as needed. For example, if the normal DITA `<task>` topic type is appropriate for a how-to publication, you can integrate the D4P formatting domain with the base `<task>` topic type and hey-presto, a Publishing task.

In short, Publishing content participates in fundamentally different business processes and supports fundamentally different business requirements than technical documentation. These differences are reflected in different publication development business processes and workflows, different approaches to authoring and revision, and different requirements for content consistency. Most publishing content is not particularly modular, certainly not below the chapter or article level. But some is (travel and nature guides, for example). Reuse within a publishing context is usually at a much larger unit of granularity than in technical documentation. It would be rare, for example, to reuse below the topic level in a typical publication.

For technical documentation the key requirements of consistency and modularity coupled with task-oriented writing practices lead to the markup designs reflected in the standard DITA task/concept/reference topic types and the bookmap map type.

For publishing content, the key requirements of flexibility and adaptability and a non-requirement for consistency or modularity (in general) leads to the markup designs reflected in the DITA for Publishers part/chapter/article/section/sidebar topic types and the publication map and publication metadata domains.

But part of the genius and magic of DITA is that these two sets of requirements and the resulting disparate markup designs are both supported equally well by the underlying DITA architecture and supporting infrastructure.

In essence, DITA lets us eat our cake and have it. Publishers can have publication models and topic types that are adapted to their specific needs yet still take advantage of all the modularity and reuse features of DITA when they are

needed. Publishers can mix and match the simple (chapter) with the sophisticated (task) and choose from a growing library of special-purpose vocabulary modules, such as the DITA Learning and Training modules for representing assessments (test questions) and formal learning objects.

A key feature of the DITA architecture is separation of concerns between publication structure and organization (maps) and publication content (topics). This separation is required in order to enable easy reuse of publication components across publications without making copies of content. The DITA map mechanism is analogous to the "virtual document" features provided by many content management systems or to the InDesign "book" feature. Strictly speaking, the use of maps in DITA is not required—you could represent an entire publication as a single topic document—but normal DITA practice is to always have maps and that practice is assumed in the DITA for Publishers design and implementation.

DITA for Publishers Vocabulary Modules Overview

The DITA for Publishers project provides map, topic, and domain specialization vocabulary modules for constructs needed by typical publishing documents.

The map specializations are:

publication map domain	Provides a rich set of topicref specializations that can be used to create DITA maps that represent almost any possible configuration of typical publication components. As a map domain, the publication map elements can be mixed with elements from other map domains, such as the Learning and Training map domain. The publication map domain has been designed to make it easy to organize publication components into submaps.
publication metadata map domain	Provides a rich set of metadata elements that reflect the needs of publishers. Publishing publication metadata tends to be more demanding than that required by technical documents. As a map domain, the publishing metadata elements can be added to any map type.
pubmap	The publication map map type integrates the publication map and publication metadata domains into a map document type. It serves both as a general publication map type and as an example of how to integrate and configure the publication map and publication metadata domains. Similar to the DITA 1.1 bookmap, but less constrained. Represents a single publication, nominally as presented in print (although may be adapted for other media formats as well). Includes publishing-specific metadata not available from bookmap, such as both 10- and 13-digit ISBNs and ISSNs. Includes the ability to represent arbitrary single pages, such as often occur in fiction books before the main content. Also includes facilities for including cover components.

The topic specializations are:

article	For articles within periodicals or article-based books. Allows nested subsection and sidebar topic types in the default shell configuration. Enables authoring of complete articles as single files.
chapter	For chapters within books. Allows nested subsection and sidebar topic types in the default shell configuration. Enables authoring of complete chapters as single files.
part	For parts within books. Provides the part title and, optionally, introductory body content. Can be configured to allow nested chapter or subsection topics but does not in the default shell configuration.
subsection	Generic subsection within articles and chapters. Enables creation of arbitrary hierarchies of titled divisions, e.g., within chapters and articles. Base for more specialized subsection types.
sidebar	Generic out-of-line topic. May contain nested subsections in the default shell configuration. Base for more specialized sidebar types.
cover	Provides a generic topic for representing publication covers. Has a specialized title element with empty content. The content of the cover topic would normally be a reference to a graphic of the cover itself.

The topic domain specializations are:

classification	Provides a generic `<classification>` element (specialized from `<data>`) that clearly identifies the metadata elements it contains as being classifying as opposed to any other type of

	metadata (such as identifying metadata). Classifying metadata normally relates its container to items in defined classification taxonomies.
enumeration	A common requirement in publishing documents is arbitrary numbering (enumeration) of document components. Sometimes this enumeration is arbitrary, meaning that it cannot be automated. Sometimes it needs to be captured for legacy or historical reasons. The enumeration domains support both the simple identification of content that serves as enumerations, allowing a processor to show or suppress the enumerations as appropriate, and author-specification of arbitrary automatic numbering. The intent of the enumeration design is to enable author-specification of arbitrary numbering rules where necessary. These domains are still somewhat experimental.
formatting	Provides elements that request or capture specific formatting effects, including line breaks, tabs, and mathematical equations represented using any combination of MathML, InDesign data, or graphics.
rendition target select attribute	Provides a global attribute intended to allow filtering and flagging based on the rendition type, e.g., PDF, EPUB, HTML, etc.
publication content	Provides elements that are unique to publications, e.g., "epigram" and "epigraph".
ruby	Provides markup identical to the HTML 5 `<ruby>` markup for annotating ideographic characters with their pronounciation, as is required for Japanese-language documents.
verse	Provides elements for representing verse as a sequence of stanzas and lines.
XML	Provides elements for identifying mentions of XML constructs.

The project's vocabulary modules also include configurations of the standard topic, concept, task, and reference topic types that integrate the various DITA for Publishers domain modules.

In addition to these base specializations (base in the sense of defining generic publishing-specific markup), the sample data includes a specialization for Shakespear plays that shows how to adapt DITA, and DITA for Publishers specializations, to the very specific task of representing plays.

Publication Component Topic Types

The five topic types article, chapter, part, subsection, and sidebar provide the basic building blocks for all publications when coupled with the standard task and glossary entry types.

All publications consist of some combination of chapters or articles, sometimes organized into parts or similar groupings. Within a chapter or article there may be a hierarchy of titled divisions (subsections) or sidebars.

The DITA for Publishers topic types `<article>`, `<chapter>`, `<subsection>`, `<sidebar>`, and `<part>`, coupled with the generic topic type `<topic>`, the glossary entry topic type `<glossentry>`, and other more-specialized topic types such as `<task>` and `<learningContent>`, can represent almost any publication with appropriate clarity and completeness without requiring additional specialization.

The content of the five topic types is identical to the base `<topic>` topic type. This provides maximum flexibility, ensuring that almost any legacy content can be captured as one of these topic types.

The main purpose of these topic types is to make it clear that a particular topic is an article or is a chapter, or at least that it started its life with that expectation. In short, authors expect to have root element types that correspond to the things they are creating as an authorial action. It also enables finding of content based on its basic role, e.g., "find all articles that contain the text 'publishing tools'".

Note that more specific structural distinctions within publications—frontmatter components, appendixes, etc.—are normally made within publication maps. A given "chapter" may act as a frontmatter section, a body chapter, or an appendix depending on how it is used in a particular publication. Of course, enterprises may decide to further specialize from these basic types.

Each of the five types may directly contain the other topic types as appropriate. Parts may contain articles or chapters. Parts, chapters, and articles may contain subsections or sidebars. Subsections may contain nested subsections or sidebars. Sidebars may contain subsections but not other sidebars (at least not directly).

An effect of this design is that an entire part or chapter or article can be authored and managed as a single XML document—there is no requirement to make each subordinate topic a separate file (DITA in general does not require it and DITA for Publishers definitely does not require it).

Keep in mind also that nested topics may be re-used from maps without first breaking them out into separate files. This means, for example, that you should be able to use by reference from another publication a sidebar originally authored as a nested topic within a chapter or article.

For publications that include glossaries, the standard DITA `<glossentry>` topic type should serve. Likewise, publications that include processes or procedures or tutorials should be able to use the standard `<task>` topic type.

Finally, any publication component for which any of these specialized types is not appropriate can use the generic `<topic>` topic type. For example, some publications include arbitrary untitled pages that play no obvious semantic or structural role, at least not one requiring codification in carefully-designed markup. In this case a generic topic with an empty title element will serve just fine (remember that one of the few hard rules in DITA is that every topic must have a title *element*—there's no rule that says that title element can't be empty).

Because publishing content is, by its nature, varied, arbitrary and largely at the whim of authors and designers, the DITA For Publishers designs must necessarily be as flexible as possible and, in general, take a much less dogmatic approach to markup than tends to be applied to the use of DITA in technical documentation, where consistency and repeatability are the driving business requirements. This means that DITA for Publishers will often explicitly allow, or even encourage, markup practices that would be anethma in the context of technical documentation but are the only way to solve certain problems in the context of Publishing.

The `<d4pCover>` topic type is intended for topics that represent graphical covers where the topic has no natural title. The `<d4pCover>` topic type provides a specialized title element that has empty content. The body of the topic is normall a reference to a graphic for the cover (e.g., the front cover or back cover of a book).

Publication Maps

The publication map and publication metadata domains provide a rich set of topic references and metadata elements for use in maps that can represent almost any existing or possible publication.

The publication map and publication metadata domains are used to create maps that represent publications of any sort. The markup is designed to enable maximum structural flexibility rather than imposing specific organizational rules. The markup design reflects the following general patterns:

- Topicrefs that represent specific components of publications, e.g., `<preface>`, have a corresponding mapref version for linking to a DITA map for that publication component (e.g., `<preface-mapref>`). While the non-mapref-specific topicref can, of course, be used to refer to submaps, the -mapref versions provide convenient markup for creating submap references and make the map author's intent clear.
- For the publication as a whole and for each submap that represents a complete publication component, there is always a root topicref element that roots the navigation tree for the publication or publication component content. This provides a consistent pattern for root maps and submaps and makes it clear which part of the map represents the content, as opposed to metadata or relationship tables. It also means that for each *-mapref element there is a corresponding topicref for use in the referenced map, so that the correspondence between map references and the referenced content is clearer to map authors.
- The domain imposes few constraints on how different kinds of topicrefs can be combined, reflecting the fact that some publications may need to do what might seem like nonsensical or non-standard things in terms of organization.
- Key definitions may be organized into a separate `<keydefs>` element or organized into a separate submap. Key definitions can also be grouped within `<keydefs>` as needed.

The publication map and publication metadata domains can be used with other map domains, such as the Learning and Training map domain or with generic maps. The pubmap map type provides a simple integration of the publication map and publication metadata domains useful for representing generic, unconstrained, publications.

DITA For Publishers Domains

The D4P vocabulary modules include a number of domains you can use to augment both map and topic types.

Domain vocabulary modules provide element types or attributes that can be mixed into map or topic types. DITA for Publishers provides a set of map domains, a set of topic domains, and a set of attribute domains.

The map domains provide publication-specific metadata (pubmetadataDomain) and publication-specific topicref types (pubmapDomain) that can be used together or separately to create new map types specifically for publications. Because these are domains they can be integrated with existing map types or they can be used to create new map types. The D4P pubmap and pub-component-map map types are examples of using the map domains to define publication-specific map types. The D4P document types also includes an example of integrating the map domains into a generic `<map>` map type. The d4p_enumerationMapDomain provides markup for use within topicrefs to define and control numbering of things, such as topic titles and lists.

The element domains provide elements for use within topics. They are:

- d4p_classificationDomain

  Provides metadata elements with the specific semantic of "classification" as distinct from any other use of metadata.
- d4p_enumerationTopicDomain

  Provides elements for capturing or controlling the number of things like topics and list items. There is a corresponding map domain, d4p_enumerationMapDomain.
- d4p_formattingDomain

  Provides elements intended to express direct formatting intent, such as line breaks, tabs, additional inline decoration types (e.g., strikethrough), and inclusion of formatter-specific snippets, such as embedded InDesign INX markup.
- d4p_pubcontentDomain

  Provides elements typical to publications, such as epigrams used in part and chapter openers.
- d4p_simpleEnumerationDomain

  Provides simple enumeration markup for simply capturing literal numbers, for example, from legacy content.
- d4p_verseDomain

  Provides markup for representing poetry and similar stanza-based content.

The attribute domains are:

- d4p_renditionTargetAttDomain

  Provides a `@props` specialization intended to specify different rendition targets such as PDF, EPUB, HTML, etc. The allowed values are not defined by the module since the set of sensible values is unbounded.

Publication Map Domains

The publication map domains provide building blocks for maps that represent publications.

If you examine any publication you will see it has two distict parts:

- The metadata that describes, names, and classifies the publication, including its title, any ISBN or ISSN numbers, copyright statements, authorship, ownership, and so on.
- The content of the publication itself: the chapters, articles, appendixes, and so on that make up the publication as a package of useful information, organized into some organizational structure (e.g., frontmatter, body, backmatter, etc.).

Both of these aspects of publications differ wildly among publishers and among publications from the same publisher. In addition, there is no general correlation between metadata requirements and content structuring requirements. Thus, two publications from the same publisher may have the same metadata requirements but different structuring requirements. Likewise two similar publications from different publishers may have the same content structuring requirements but different metadata requirements.

Thus it would almost never be the case that a single map type that defined both metadata structures and content structures would meet anybody's requirements very well.

Thus, rather than defining a single "publication map" map type that tries to cover all possible metadata and structure requirements, DITA for Publishers provides two separate map domains, one for metadata (pubmetadata domain) and one for content structuring (pubmap domain).

As domains these two domains can be used together or separately. They can be integrated into existing map types or used to construct completely new map types. This maximizes the flexibility of the mechanism.

DITA for Publishers includes three sample map types that demonstrate using these domains:

- `<map>`

  The DITA for Publishers map topic type simply integrates the pubmap and pubmetadata domains with the generic DITA `<map>` map type, making the D4P metadata and topicref types available in a largely unconstrained way.
- `<pubmap>`

  Integrates the pubmap and pubmetadata domain to create a more focused publication map.
- `<pub-component-map>`

  Also integrates the pubmap and pubmetadata domains, but for the creation of publication subcomponents, such as part maps or chapter maps.

For most publishers the generic pubmap and pub-component-map types will be sufficient to represent most, if not all, publications. You can, of course, use constraint modules to make the base map types more constrained. You can also use either or both modules as the basis for further specialization or simply define your own domains as needed.

Topic Domains

The topic domains provide markup commonly needed in publications that would not be needed or wanted in most technical publications.

The standard DITA markup design is driven by the normal business requirements for technical documents, which include the avoidance of author-specification of layout and formatting details, including numbering. Thus DITA has a minimum number of "arbitrary formatting" elements. Publishers are driven by almost completely opposite business requirements.

Publishers must be able to express arbitrary, even non-sensical, formatting intent on behalf of authors. Publishers must also be able to do things like numbering in flexible and ideosyncratic way. Publishers must be able to capture legacy content details so they can be reproduced or searched on.

The DITA for Publishers topic domains attempt to address these requirements in a general way and show a path toward more specific markup as may be needed by specific publishers or for specific publications.

Using these domains it should be possible accurately capture most or all of the essential formatting details within publication content, such as numbering of headings, paragraphs, and list items, line breaks, inline font and decoration changes, and so on. It is not the intent of these domains to capture the full details of each page in a legacy document--that is simply not possible in a general XML format.

In addition to formatting, these domains also provide semantic elements for things that are typical of publications and not usually found in technical documents, such as poetry and epigraphs.

The classification domain provides a general-purpose specialization of `<data>` that expresses the semantic of "classifying metadata" as distinct from any other form of metadata. The idea is that the `<classification>` element contains metadata elements that bind content to specific classifiers, usually in terms of some separately-defined taxonomy.

The point of having a specific `<classification>` element is twofold. First it makes it clear to authors and markup designers that "classification goes here". It also enables more focused classification-based searching by providing a consistent container for metadata that holds classification details. This can make it easier to configure things like faceted search. Otherwise there is no magic to it.

The general use intent of the classification domain is that you will define additional domain modules that provide the specific classification tagging you need that reflects the specific taxonomies or classification schemes you use.

However, you can also just use generic <data> elements with appropriate @name values and content or @value values without specializing.

Classification Domain

The classification domain provides metadata specifically for classifying topics or maps, typically in terms of a specific taxonomy.

The classification domain defines the metadata element <classification>, which is intended to hold metadata that serves to classify a topic or map in terms of one or more taxonomies. It's primary purpose is simply to provide a clear and convenient place to hold classification metadata or to enable the definition of more precise content models for classifying metadata such that the document type rules can require specific metadata elements.

Enumeration Domains

Provides elements for simply identifying enumerations within content (simple enumeration domain) or for defining automatic enumeration "streams".

The enumeration domains provide markup for both identifying enumerations within content (simple enumerations) and for defining automatic enumeration "streams".

The simple enumeration domain (d4p_simpleEnumeration) provides the element type <d4pSimpleEnumerator>, a specialization of <data>. Use <d4pSimpleEnumerator> to identify the occurrence of things like chapter and section numbers within content where you need to preserve the number for legacy purposes or to reflect author intent but where it may be necessary to suppress the number in different use or publishing contexts.

The problem with numbers of course is that they reflect a specific use of a topic or element within a specific packaging context. Thus, what is Chapter 2 in one book may be Chapter 3 in a different publishing context (or even a later revision of the same publication) or may not be usefully numbered at all.

Ideally, all numbers would always be generated automatically. However, in a Publishing context that is not always practical. Legacy documents may have ideosyncratic numbers that cannot be automated. New documents may require numbering that is automatable but unique to that particular document.

The <d4pSimpleEnumerator> allows identification of numbers in a way that clearly distinguishes the number from surrounding content, allowing it to be formatted appropriately, including suppressing it altogether. The <d4pSimpleEnumerator> is a specialization of <data>, which means it is semantically metadata and will be hidden by default. The DITA for Publishers project provides Toolkit plugins that extend the HTML and PDF transforms for the simple enumeration domain that unhide the <d4pSimpleEnumerator> elements, making them visible by default.

The two enumeration domains d4p_enumerationMapDomain and d4p_enumerationTopicDomain provide support for author-specified automatic numbering that is arbitrary. The enumeration map domain provides elements for defining enumeration "streams". The enumeration topic domain provides elements for generating numbers within a specific stream.

Important: The map and topic enumeration domains are experimental and not yet fully implemented.

Formatting Domain

The formatting domain provides elements for producing specific formatting effects that are either arbitrary (not connected to any particular semantic or structural aspect of the content) or simply required to meet specific publication requirements. These elements should be used only when necessary but when they are necessary they are available.

The elements in this module are:

<art>	Represents an abstract piece of "art". Holds a reference to one or more graphic objects (<image>) (normally different formats of the same base image, such as a high-resolution TIF and low-resolution JPEG). It may have an "art title", which labels the art as a re-usable object (e.g., when displayed in a CMS system as a standalone object). It may also contain classifying metadata.
<b-i>	"Bold italic". Allows a single element to represent the combination of <b> and <i>, which is useful for conversion from flat word processing formats.

<b-sc>	"Bold small caps". Allows a single element to represent the combination of <b> and <sc>, which is useful for conversion from flat word processing formats.
** **	Represents a hard return (as for HTML element). Can be used for example in titles to indicate a hard break point.
<eqn_block>	Represents a block equation. Same content as for <eqn_inline>.
<eqn_inline>	Represents an inline equation. Can contain any combination of MathML markup, InDesign INX markup, or references to graphics (<art>).
<frac>	Represents a simple mathematical fraction. Enables the rendering of fractions without the use of full MathML markup.
<inx_snippet>	Holds a snippet of InDesign CS3 interchange XML. This element is intended to be a fallback for things like equations that simply cannot be easily represented in any other way for printing purposes. Should be used only when absolutely necessary. INX snippets can only be rendered directly by InDesign.
<linethrough>	Line-through or "strikeout". Text with a line drawn through it as is used for strike-outs.
<roman>	"Roman" (not italic, bold, or small caps). Use to indicate text that is not highlighted, usually within the context of a paragraph-level element that is normally highlighted in some way, such as paragraphs that are normally rendered in italic.
<sc>	"Small caps". Represents a phrase in small caps.
<tab>	Represents a horizontal tab character. Literal tabs cannot be reliably entered in XML data, so this element allows the capturing of tabbed content. Note that the tab stops applied to the tabs is a function of the rendition applied.

The formatting domain includes the standard MathML markup for use with <eqn_inline> and <eqn_block>. The ability to render MathML depends on the rendering system available. MathML rendering is available for HTML and XSL-FO-based outputs. Rendering MathML into InDesign requires commercial tools.

Publication Content Domain

The publication content domain provides general-purpose content elements that are specific to publications.

The publication content domain currently defines two element: <epigram>, a pithy saying, usually humorous, and <epigraph>, a brief quotation used to introduce a piece of writing. See *http://www.wsu.edu/~brians/errors/epigram.html*.

Rendition Target Attribute Domain

Provides a @props specialization for flagging and filtering based on output type (e.g., PDF, EPUB, etc.).

The rendition target attribute domain (d4p_renditionTargetAttDomain) provides a single attribute, @d4p_renditionTarget, that is intended to define the applicability of elements to specific output types (renditions), e.g., "pdf", "epub", "html", etc. The set of allowed values is not defined in the domain as the set of sensible values is unbounded and very dependent on your local processing requirements.

Ruby Domain

Provides markup based on the HTML 5 <ruby> elements for annotating ideographic characters as is done in Japanese-language documents.

The ruby domain provides a single top-level element, @ruby, which then holds some combination of phrase-level content (the text to which the ruby applies), and some combination of <rb>, <rp>, and <rt> elements. These elements correspond directly to the HTML 5 and HTML 4 @ruby markup (the main difference being that in HTML 5 the <rb> (ruby base) element is not used).

For example, Japanese content with a ruby would be marked up like so:

```
<p> 探険船シビリアコフ号の北氷洋航海中に撮影されたエピソード映画の中に、一頭の<ruby>
<rb>白熊</rb>
<rp>（</rp>
<rt>しろくま</rt>
```

```
<rp>) </rp>
</ruby>を射殺し、その子を生け捕る光景が記録されている。</p>
```

For HTML output the markup is output exactly as tagged in the DITA source and most browsers will render the ruby text above the base text. When that rendition is not provided, then the ruby text is given after the base text within parantheses.

Verse Domain

The verse domain provides markup for representing poetry, songs, and other line-oriented content organized into stanzas.

The <verse> element contains any combination of <verse-line> and <stanza> elements. The <verse> element specializes from <lines>, so all white space within the <verse> element is significant by default.

A <verse-line> represents a single line of verse and a <stanza> represents a set of lines:

```
<verse><stanza
><verse-line>'Twas brillig and the slithy tothes</verse-line>
<verse-line>Did gyre and gymbal in the wabe</verse-line>
<verse-line>All mimsy were the borogroves</verse-line>
<verse-line>And the mome raths outgrabe.</verse-line></stanza></verse>
```

Which might be rendered like this:

'Twas brillig and the slithy tothes
Did gyre and gymbal in the wabe
All mimsy were the borogroves
And the mome raths outgrabe.

XML Domain

The XML domain provides phrase-level elements for identifying mentions of XML components. It is convenient for documents that discuss XML markup (such as this one).

The XML domain provides the following element types:

<xmlelem> Identifies a mention of an XML element type (tagname):

The markup:

```
<xmlelem>my-tagname</xmlelem>
```

Produces: <my-tagname>

<xmlatt> Identifies a mention of an XML attribute:

The markup:

```
<xmlatt>my-attname</xmlatt>
```

Produces: @my-attname

<textent> Identifies a mention of an XML text entity:

The markup:

```
<textent>my-text-entity</textent>
```

Produces: &my-text-entity;

<parment> Identifies a mention of an XML parameter entity:

The markup:

```
<parment>my-parameter-entity</parment>
```

Produces: %my-parameter-entity;

<numcharref> Identifies a mention of an XML numeric character reference:

The markup:

```
<numcharref>2014</numcharref> (em-dash)
```

Produces: ߞ (em-dash)

Integrating D4P Modules Into Document Type Shells

To use the vocabulary modules with your own map and topic types you must update your document type shells to include the DITA for Publishers vocabulary modules.

Modules are "integrated" with map or topic types by including references to them in the document type shells for those map or topic types.

The DITA For Publishers includes sample document type shells for all of the map and topic types it defines as well as some examples of integrating DITA for Publishers domains and types with standard topic types (such as the generic `<topic>` topic type).

Integrating modules into document type shells is an entirely mechanical process--you don't have to understand DTD or XSD syntax in order to do it, you just have to follow the steps very precisely. Of course, because you are dealling with fiddly syntax, it helps to understand how the syntax works in order to fix errors you may inadvertently introduce.

Because DITA for Publishers defines so many different topic types it uses a not-strictly-conforming technique to make it easy to include all the topic domains into a topic shell. This is the "common" files in the doctypes/common directory. These common files simply collect in one file all the entity declarations and references required by the DITA specification. They are fulling conforming individually but the DITA 1.2 spec does not actually allow this form of common declaration file.

However, all the DITA for Publishers shell document types can be made strictly conforming simply by copying the contents of each common file at the point of reference to each of the common files. So I'm not too worried about it. My intent is not to subvert the DITA requirements in any way, but simply to make the job of maintaining the various shells easier. Once the D4P vocabulary design has become more stable I will likely remove the use of the common files, making the shells strictly conforming.

Integrating Attribute Domains

Attribute domains are defined in a single .ent file that defines a single parameter entity with the entity declaration and a single general text entity that provides the `@domains` attribute contribution for the module.

To integrate an attribute domain into a DTD document type shell, follow the steps shown here. If you do not already have your own document type shell for the topic or map type you want to integrate with, copy the closest starting point (e.g., one of the D4P .dtd files) and then modify that. [Add reference to specialization tutorial once it is updated to DITA 1.2.]

1. Insert a declaration for and reference to the attribute domain's .ent file.

   Look for a comment that looks like this:

   ```
   <!-- ============================================================= -->
   <!--                    DOMAIN ATTRIBUTE DECLARATIONS              -->
   <!-- ============================================================= -->
   ```

   After this comment, add an entry that looks like this (e.g., for the d4p_renditionTarget attribute domain):

   ```
   <!ENTITY % d4p_renditionTargetAtt-d-dec
     PUBLIC
   "urn:pubid:dita4publishers.sourceforge.net:doctypes:dita:modules:d4p_rendi
   tionTargetAttDomain:entities"

   "d4p_renditionTargetAttDomain.ent"
   ```

```
>
%d4p_renditionTargetAtt-d-dec;
```

2. Add the domain's parameter entity the `%props-attribute-extensions;` parameter entity.

   Find a comment that looks like this:

   ```
   <!-- ============================================================= -->
   <!--                      DOMAIN ATTRIBUTE EXTENSIONS              -->
   <!-- ============================================================= -->
   ```

   Immediately following that comment should be a line that looks like this:

   ```
   <!ENTITY % props-attribute-extensions "" >
   ```

   Between the double quotes (or before the ending double quote if there is already a value in the declaration, add a reference to the parameter entity defined in the .ent file included in the previous step. The parameter entity *should* be named "*%attributeName*Att-d-attribute" but you will need to look in the .ent file to make sure. The result should look like this:

   ```
   <!ENTITY % props-attribute-extensions "%d4p_renditionTargetAtt-d-
   attribute;" >
   ```

   Note the "%" at the start of the entity name (this is a parameter entity reference open character) and the ";" at the end (this an entity reference close character).

3. Add a reference to the domain's `@domains` attribute contribution entity to the "included-domains" general entity declaration.

   Look for a comment like this:

   ```
   <!-- ============================================================= -->
   <!--                      DOMAINS ATTRIBUTE OVERRIDE               -->
   <!-- ============================================================= -->
   ```

   And then a declaration like this:

   ```
   <!ENTITY included-domains
                           ""
   >
   ```

   Your shell may already have things inside the double quotes.

   Within the double quotes add a reference to the domains attribute component entity for the domain. This *should* be named like "&*attributeName*Att-d-att;" but you will need to look in the .ent file to make sure. The result should look something like this:

   ```
   <!ENTITY included-domains
                           "
                            &d4p_renditionTargetAtt-d-att;
                           ">
   ```

   The line breaks within the declaration and within the double quotes don't matter. It is convenient to put each entity reference on its own line so its each to update the declaration as you add or remove domains.

   Note the "&" at the start of the entity name (this is a general entity reference open character) and the ";" at the end.

Your shell should be ready to use. How you test it depends of course on your working environment. Deployment and testing in specific tools is beyond the scope of this document.

DITA For Publishers Vocabulary Reference

This section provides reference entries for each of the DITA for Publishers element types and attributes.

This section is an alphabetical reference by element type and attribute name for all the DITA for Publishers vocabulary modules.

The DITA For Publishers Word to DITA and DITA to InDesign Tools

This part documents the Word-to-DITA and DITA-to-InDesign transformation frameworks.

The Word-to-DITA and DITA-to-InDesign transformation frameworks enable the use of DITA with Word as the primary authoring and editorial workflow vehicle and InDesign as the primary print production technology.

These two frameworks reflect the fact that for many publishers it is simply impractical or impossible to use an XML-first or XML-only editorial process, especially when dealing with external authors and subject matter experts. For good or ill, Microsoft Word is the form most authors can or will use. Likewise, InDesign is almost universally accepted as the tool for producing high-quality typeset publications. Without ways to integrate these two technologies into a DITA-based environment, the use of DITA (or XML in general) is a non-starter for most Publishers.

The technologies documented in this part are *frameworks*, meaning that they require some degree of configuration and customization to be useful in a specific context—you cannot simply take an arbitrary Word document and expect any transform to give you a useful XML result. Likewise, the use of InDesign (or any desktop publishing system) requires a mapping to specific styles and layouts that will always be specific to your documents.

Given that configuration is always required, these frameworks have been designed to make the configuration and, where necessary, customization process as easy as possible.

The Word-to-DITA transformation can be configured largely or entirely (depending on your requirements) using an XML configuration file to map Word styles to DITA markup structures. It can handle going from Word documents to single topics or to complete systems of maps and topics.

The DITA-to-InDesign transformation is configured through a simple-to-modify XSLT module that maps DITA elements in context to InDesign style names.

When requirements are not too extreme it is possible to get a demonstrable end-to-end-with-DITA-in-the-middle system going in less than a day, including defining custom vocabulary modules.

These frameworks don't do everything you may want to have and are not intended to compete with or replace commercial tools. They are intended primarily to get you started and to provide a starting point for further development and extension.

For example, Word-based authoring can be made more effective and reliable by using Word macros to guide authors in styling. Likewise, InDesign scripting can go a long way toward automating some repetitive layout tasks by taking advantage of the additional structure provided by base markup. There are a number of commercial products that offer support for authoring XML in Word or generating XML from styled word, such as the Inera eXstyles product (*www.inera.com*).

Finally, the production of complete, print-ready publications from XML through InDesign is an inherently challenging process. The DITA For Publishers DITA-to-InDesign framework does not attempt to solve that problem. It is intended primarily to support processes where designers must be directly involved in the layout, such as magazine production, or where there is no attempt to completely automate composition, but simply to provide the input to an more traditional manual layout process.

To fully automate production from XML the Typefi product is the best solution (*www.typefi.com*). Typefi enables up to 100% automation of InDesign-based production where book design is consistent across titles, e.g., for series publications like travel guides, journals, and so on. Typefi provides a level of sophistication that would be impossible (and pointless) to replicate. Typefi is a for-cost product but it offers a tremendous value when automated composition is a requirement.

The Word-to-DITA Transformation Framework

The Word-to-DITA (Word2DITA) transformation framework enables the reliable generation of maps and topics form styled Word documents.

The Word-to-DITA transformation framework provides a general facility for converting styled Microsoft Word documents into DITA maps and topics. It is intended primarily to support ongoing authoring of DITA content using Microsoft Word, where all authoring is done in Word and DITA maps and topics are generated from the Word on demand. It does **not** provide a way to go from DITA back to Word (although that would be technically possible, if not trivial).

The Word-to-DITA transformation is **not** intended to support general data conversion from arbitrary Word documents to DITA. It requires at least some amount of consistent styling. However, it may still produce useful starting points from lightly-styled documents. Try it and see. Because the output of the transform can be quite complete, it may be most effective to do data cleanup in Word (that is, applying appropriate styles) and then use the transform to generate more-or-less ready-to-use maps and topics, rather than generating DITA content that needs significant rework to be usable.

The Word-to-DITA process can do any of the following:

- Generate a single DITA topic document from a single Word document
- Generate a single DITA map document and one or more topic documents from a single Word document
- Generate a tree of maps and one or more topic documents from a single Word document.

Transformations are from Word documents using a specific set of named styles to DITA documents of any type. The transformation is defined entirely or mostly through a declarative style-to-tag map that defines how each Word paragraph and character style maps to specific DITA structures.

The declarative style-to-tag map makes it quick and relatively easy to set up and maintain conversions. As long as the style-to-tag mapping is sufficient no XSLT programming is required.

If the style-to-tag map is not sufficient then you can extend the base Word-to-DITA transform using XSLT. Some reasons to extend include:

- Handling complex structures that cannot be expressed with declarative mapping (usually deeply-nested structures)
- Implementing custom rules for assigning element IDs.
- Implementing custom rules for constructing map and topic filenames.

The Word-to-DITA transform can be used either through the DITA Open Toolkit or as a standalone XSLT transformation (for example, to embed it in a CMS-managed tool chain). While it is packaged as an Open Toolkit plugin, it has no dependencies on any Toolkit components—it just uses the Toolkit's processing framework to make it convenient to apply the transform.

Getting Started With The Word2DITA Transform

How to set up an Ant task to convert a Word document to DITA using the Open Toolkit

The Word2DITA transform is packaged as an Open Toolkit plugin (see *Generating DITA from Documents (Word-to-DITA Transformation Framework)* on page 26). This makes it easy to apply the transform to Word documents using an existing Open Toolkit installation.

The basic steps to follow are:

1. Deploy the DITA for Publishers Toolkit plugins to your Open Toolkit (see *Installing the Toolkit Plugins* on page 12).
2. Create a file named `build.properties` in your home directory (e.g., `/Users/ekimber`) and in that file put a line like this:

   ```
   dita-ot-dir=c:/DITA-OT
   ```

   Where the part in bold reflects the location of the Toolkit on your machine.
3. Copy the `word2dita/all_defaults` directory to a convenient place. This will be the starting point for your word-to-DITA transformation configuration.
4. Edit the file `dita-ot-run-word2dita.xml` and change this line:

   ```
   <property name="args.input"
     location="${myAntFile.dir}/
   ```

```
word2dita_single_doc_to_map_and_topics_01.docx"
/>
```

To reflect the filename of the Word document you want to convert.

5. Edit your Word document and make sure that the first paragraph in the document is styled with the "Title" paragraph style—you may need to add a new paragraph. This paragraph defines the map title and signals the generation of the root output map. This setup is required by the default style-to-tag mapping.
6. Run the `dita-ot-run-word2dita.xml` with Ant:

```
ant -f dita-ot-run-word2dita.xml
```

If you've set the dita-ot-dir Ant property correctly then it should run and you should get some output. You should get a map and some number of topics, one for each Heading 1, Heading 2, Heading 3, and Heading 4 paragraph in your Word document.

The Toolkit log will include messages from the Word-to-DITA transform, which will report unmapped paragraph and character styles. By default, any unmapped paragraph style is mapped to `<p>`, so you will usually get valid, if not ideal, output from the default mapping.

Once you have verified that you can get *something* from the transform you are ready to start configuring the style-to-tag mapping to reflect your specific requirements and Word documents. You will likely find that you also need to refine how your documents are styled so that they map most effectively.

Finally, remember that the Word-to-DITA process is a work in progress and there is always room for improvement. Please report any bugs or feature requests to the DITA for Publishers bug tracker on SourceForge (*https://sourceforge.net/tracker/?group_id=262530&atid=1259341*). The Word to DITA transform has been used in production by several companies but for a relatively narrow set of requirements, so I cannot claim to have tested it broadly.

Remember too that the Word to DITA framework is open source, which means I welcome fixes and enhancements from the community.

Generating DITA from Word Using the Toolkit Plugin

How to use the Word-to-DITA Toolkit Plugin to generate DITA content from Word documents.

To use the transformation as a Toolkit plugin simply deploy the DITA for Publishers Toolkit plugins to your Open Toolkit directory. The Word-to-DITA transformation type is "word2dita".

Required Parameters

args.input

The absolute filename and path of the DOCX file to process.

w2d.style-to-tag-map

The absolute filename and path of the style-to-tag mapping to use.

Optional Parameters

output.dir

Specifies the absolute path of of the directory to contain the generated output.

w2d.clean.output.dir

When set to "true" deletes any files in the output directory before running the generation.

w2d.filter.br

When set to any value (e.g., "true"), filters out literal break characters, which are otherwise preserved in the DITA output as `<br>` elements.

w2d.filter.tabs

When set to any value (e.g. "true"), filters out literal tab characters, which are otherwise preserved in the DITA output as `<tab>` elements.

w2d.root.output.filename

The filename to use for the root output file (the root map or root topic depending on how your mapping is set up), e.g. "rootmap.ditamap".

w2d.word2dita.xslt

Specifies the name of the XSLT transform to use to convert the DOCX file to DITA. Default is the `docx2dita.xsl` transform in the word2dita Toolkit plugin.

w2d.debug.xslt

Turns on debugging of the XSLT transform. Default is "false".

Setting Up an Ant Script for Running the Word-to-DITA Transform

How to set up an Ant task to run the Word-to-DITA transform against DOCX files.

You can set up an Ant script to run the transform against a specific Word document. A typical script looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Single Word Doc to Map and Topics 01"
default="transformMyDoc">

  <property file="build.properties"/>
  <property file="${user.home}/.build.properties"/>
  <property file="${user.home}/build.properties"/>

  <!-- This property should be set in one of the included files above
       to reflect the location of Toolkit on your machine:
    -->
  <property name="dita-ot-dir" location="c:\DITA-OT1.5"/>

  <dirname property="myAntFile.dir" file="${ant.file}"/>

  <tstamp/>

  <target name="transformMyDoc">

    <property name="word.doc"
      location="${myAntFile.dir}/
word2dita_single_doc_to_map_and_topics_01.docx"/>

    <basename  property="doc.base.name" file="${word.doc}" suffix=".docx"/>

    <ant antfile="${dita-ot-dir}/build.xml" target="dita2word2dita">
      <!-- Set this to the filename of the DOCX file to be transformed: -->
      <property name="args.input"
        location="${word.doc}"/>

      <!-- Change w2d.style-to-tag-map to point to your style-to-tag map
document: -->
      <property name="w2d.style-to-tag-map"
          location="${myAntFile.dir}/style2tagmap.xml"/>


      <property name="w2d.root.output.filename"
          value="${doc.base.name}.ditamap"/>

      <!-- Specify output.dir to set to the output directory to contain the
generated DITA files: -->

      <property name="output.dir" location="${myAntFile.dir}/../dita"/>
      <property name="w2d.clean.output.dir" value="true"/><!-- Clean output
directory before doing generation -->
      <property name="w2d.debug.xslt" value="true"/><!-- Turn XSLT debug
messages on -->
```

```
    </ant>
    <ant antfile="${dita-ot-dir}/build.xml" target="dita2xhtml">
      <property name="args.input"
        location="${myAntFile.dir}/../dita/${doc.base.name}.ditamap"/>
    </ant>
  </target>

</project>
```

The text in bold is what you would change to reflect your documents.

Note the initial `<property>` elements that import a file named `build.properties`. This approach lets you put this script anywhere and use the build.properties file to set the location of your Toolkit, defined in the Ant property *dita-ot-dir*. (See *Getting Started With The Word2DITA Transform* on page 41 for more details on this one-time setup task.)

If you name this file `build.xml` in an appropriate directory (e.g., the same directory as the Word document), you can run this Ant script from the command line like so:

```
c:\workspace\myworddoc\> ant
```

You can also specify the Word document as a command-line parameter:

```
c:\workspace\myworddoc\ > ant -Dword.doc=c:/workspace/anotherworddoc/some-
document.docx
```

TODO: It should be possible to use Ant to convert a set of documents in one go but I haven't worked out the details for that yet.

If you want to get really sophisticated you can extend your Ant script to run the Toolkit against the DITA files you just generated, for example, to generate HTML as a way to validate the generated files. To do that you would add this `<ant>` element after the `<ant>` element that runs the Word-to-DITA transform:

```
    <ant antfile="${dita-ot-dir}/build.xml" target="dita2xhtml">
      <property name="args.input"
        location="${myAntFile.dir}/../dita/${doc.base.name}.ditamap"/>
    </ant>
```

You can, of course, add any other normal Toolkit arguments you might want to add.

Generating DITA From Within OxygenXML

You can transform Word documents from within OxygenXML without using the Open Toolkit

OxygenXML has two handy features that make transforming Word documents into DITA convenient, especially as you are defining the style-to-tag mapping or implementing custom XSLT extensions:

- OxygenXML can open Zip archives in its Archive Browser. As DOCX files are just Zip archives, it means you can open Word DOCX files in Oxygen and get to the individual files inside very easily.
- OxygenXML can run XSLT transforms and, in particular, can run them against files within Zip packages.

In addition, you can use Oxygen's built-in Open Toolkit to manage resolution of the XSD schema for style-to-tag mapping documents simply by deploying the DITA for Publisher's Toolkit plugins to Oxygen's Toolkit instance (see *Installing the Toolkit Plugins* on page 12).

You can use Oxygen's normal Toolkit transformation scenario to run transforms against Word documents. Simply set the transtype Ant parameter to "word2dita" and specify the Word document as the input file.

You can also transform Word documents directly in Oxygen, which is handy for testing and development of your style-to-tag mapping.

To run the transform in Oxygen, do the following:

1. Open the DOCX file you want to transform in Oxygen's Archive Browser
2. Navigate to the file `word/document.xml` within Archive Browser and open it in the editor (double click on the filename).

   All Word documents have a file named `document.xml`, which contains all the paragraph content for the document. This file is the input to actual Word-to-DITA transform.
3. Open the file `net.sourceforge.dita4publishers.word2dita/xsl/docx2dita.xsl` from the Toolkit plugin, either as deployed to your Toolkit or from the DITA for Publishers distribution package (the transform has no dependencies on any other Toolkit components so you can run it standalone).
4. From the docx2dita.xsl file, select the **Configure Transformation Scenario** button from the Toolbar and select **New** to create a new transformation scenario.
5. For the XML file select the `document.xml` file you opened from the DOCX archive.
6. Select the **Parameters** button to open the Parameters dialog and set the following parameters:
   outputDir

   The URI of the output directory to hold the generated files.

   styleMapUri

   The URI of the style-to-tag map document to use for the transform.
7. Save the scenario and select **Transform now** to run the transform.

You should get some output. Use Oxygen's **File->Open** dialog to open the generated file (map or topic).

You can now rerun the transform at will and see the updated files in the editor.

Style to Tag Mapping

The docx2dita transform is driven by a configuration file, the style-to-tag map. The style-to-tag map enables an entirely or almost-entirely configuration-driven transformation of styled Word documents into DITA-based XML, specialized or not. Through careful design of Word templates it is possible to quickly configure the transformation to produce good XML documents.

The style-to-tag map consists of a sequence of `<style>` elements, where each style element defines the mapping for a paragraph, character, list, or table style to the appropriate DITA XML structure. The mapping document may also include an initial `<documentation>` element that can hold documentation in any markup language (its content is ignored and not validated), as well as `<output>` elements that define the public and system IDs to use for any generated map or topic documents.

The basic rules and constraints for the mapping are:

- The result document always consists of a single root topic or map. The first non-skipped and non-root-topic-prolog paragraph in the Word document must be of structure type "topicTitle", "map", or "mapTitle" and level "0" (zero).
- Paragraph styles that map to topic titles and contained elements (within topic bodies) must map to exactly one fixed level. Thus, to have two levels of subsection topics, you must have two separate paragraph styles, one that maps to level 1 and one that maps to level 2, even though both map to `<subsection>` topic types.
- Character style mapping is flat within a paragraph: there is currently no general way to infer multiple levels of inline markup from a flat sequence of character runs (this is more a limitation of Word's character styles than of the mapping mechanism). In theory it would be possible to use marker characters or character styles or, more reliably, embedded XML elements, to capture more complex inline structures.
- Each mapped style effectively says "I'm this type of structure, I go in this part of the topic, and (if necessary), I'm contained by this containing element." That, plus other details as needed for specific types of target elements, is sufficient to enable accurate creation of DITA XML from appropriately-styled Word documents.
- Common containment is determined by adjacency of similar container types and levels. For example, within a topic body, each sequence of adjacent paragraphs with the same `@containerType` and `@level` value will be grouped together within the same result element as determined by the `@containerType` value.

Each style mapping has the following required attributes:

@styleName

The Word style name. This is the value in the `@w:val` attribute of the `<w:style>/<w:name>` element whose `@w:styleId` value matches the style ID specified by a given paragraph, text run, list item, or table. This is the display name of the style as shown in the various style-related dialogs in Word. The style name is not case sensitive, so "Heading 1" and "heading 1" identify the same style. Note that styles may have associated aliases. Mapping to aliases is not supported as of version 0.9.16.

If you specify `@styleName` you should not specify `@styleId`. If you specify both then `@styleName` is used.

@styleId

The Word style ID. This is the value in the `@w:val` attribute of the `<w:*Style>` element associated with given paragraph, text run, list item, or table. Note that this value is not the same as the display name of the style. For example, spaces and underscores in style names are removed to form the style ID. In general, you can construct the style ID for a given style by simply removing all spaces from the style's display name.

If you specify `@styleId` you should not specify `@styleName`. If both are specified, `@styleName` is used.

@structureType

Indicates the type of target structure the Word construct represents. Possible values are:

topicTitle	Paragraphs that start new topics and provide their titles. The style mapping must also indicate the topic type, prolog type, body type, and topic nesting level.
map	Paragraphs that start new maps. Normally used only to generate the root map for the result.
mapTitle	Paragraphs that start new maps and provide their titles.
topicHead	Paragraphs that generate new `<topichead>` elements in the result map and provide their titles.
topicGroup	Paragraphs that generate new `<topicgroup>` elements in the result map. Note that topic groups never have titles.
section	Paragraphs that start new sections and, optionally, provide their titles. The style mapping must specify the section tag name and how the section title is determined (`@spectitle`).
dt	A paragraph that is the definition term part of a definition list entry.
dd	A paragraph that is the definition description part of a definition list entry. Must immediately follow a "dt" structure type paragraph.
skip	The paragraph is completely ignored by the transform. Skipped paragraphs are ignored for the purpose of determining adjacent paragraphs when determining common containers. This means, for example, that a sequence of paragraphs mapped to the same level 1 container, interrupted by a skipped paragraph, and followed by more paragraphs in the same level 1 container will be in a single container instance in the result.
xref	Represents a cross reference. Intended to map to `<xref>` or a specialization of `<xref>`.
shortdesc	A paragraph that serves as the shortdesc for its containing topic.

@topicZone

Indicates which part of the target topic the result element belongs in. Possible values are:

titleAlts	Provides one or more title alternatives
shortdesc	Provides the short description for the topic.
prolog	Contributes to the topic prolog.
body	Contributes to the topic body as a block-level element.
inline	An inline element contained by some other generated element.
topicmeta	An element that goes in the topicmeta container of a map or topicref element.

`@containingTopic` Indicates which topic should contain the element: "root" or "current". When `@containingTopic` is "root", the paragraph will be included in the root topic, regardless of where it occurs within the input Word document. This is intended primarily for prolog components.

`@containerType` For components that imply a container, such as list items, specifies the tag name of the containing element, e.g. "ul" for unordered list items. The container type value, along with the `@level` value, is used to group sequences of adjacent paragraphs together. All adjacent paragraphs with the same `@containerType` and `@level` value will be grouped together under a single instance of the specified container.

`@tagName` Specifies the XML tag name for the result element generated directly from the Word component.

`@topicType` For structure type "topicTitle", specifies the topic element tag name.

`@prologType` For structure type "topicTitle", specifies the prolog element tag name.

`@bodyType` For structure type "topicTitle", specifies the body element tag name.

`@level` Indicates the nesting level for a given topic or other containing structure. Styles within the Word document are always specific to a given nesting level (otherwise there is no reliable way to map a flat list of paragraphs into a hierarchy of XML elements). For structure type "topicTitle", level "0" indicates the root topic for the result document. Note that the level hierarchy for topic- and map-creating paragraphs is distinct from the level hierarchy for paragraphs that contribute to topic bodies (topicZone="body"). For body paragraphs, the level is relative to the body itself. E.g., level="1" would be for first-level list items, level="2" for second-level list items, and so on.

`@spectitle` For section structure types, indicates how the section title is constructed. A section-generating paragraph can also provide the section title text for use in a `<title>` element or the section title can be provided by the paragraph following the section-generating paragraph. The possible values for `@spectitle` are:

#toColon	The paragraph is expected to have initial text terminated by a colon and whitespace (": "). The section's `@spectitle` attribute is set to the value of the text up to but not including the colon. The text following the colon and whitespace is used as the first paragraph of the section, if any.
#toEmdash	The paragraph is expected to have initial text terminated by an em dash ("—") and optional whitespace. The section's `@spectitle` attribute is set to the value of the text up to but not including the dash. The text following the dash is used as the first paragraph of the section, if any.
#toPeriod	The paragraph is expected to have initial text terminated by a period and whitespace (". "). The section's `@spectitle` attribute is set to the value of the text up to but not including the period and whitespace. The

	text following the period is used as the first paragraph of the section, if any.
literal text	The text is used as the value of the section's `@spectitle` attribute.
`@useContent`	Indicates whether all the paragraph's content should be used ("mixed"), only element content should be used ("elementsOnly"), or only the text content, with all text runs included by any mappings ignored ("textOnly"). The "textOnly" value is intended for use in generating attribute values, e.g., values for the `@value` attribute of `<data>` elements. When the value is "elementsOnly", only text runs with character styles will be used in the result XML. All other content will be ignored. This is useful for extracting metadata fields from text content where the text itself is not needed in the XML (because it will be generated at rendition time, for example).
`@baseClass`	Specifies the DITA base class for the element to which the component is mapped, e.g. " topic/data ", " topic/author ", etc. This allows the transform to put the result element in the correct location, especially in topic prologs where elements can only occur in a specific place in the sequence of prolog elements. The value is as for the DITA `@class` attribute, including leading and trailing whitespace. The values recognized are a function of the specific docx2dita transformation used. In general, values should be the most general DITA-defined classes, but extensions to the base transform may expect or recognize more specialized values. Only needs to be specified for elements that must occur in a specific place within a given topic zone in the result document.
`@putValueIn`	For components that map to `<data>` or a specialization of `<data>`, indicates where to put the data value: the `@value` attribute ("valueAtt") or content ("content").
`@dataName`	For components that map to `<data>` or a specialization of `<data>`, the value to use for the `@name` attribute. If not specified, the `@name` attribute is not constructed in the result document.
`@format`	For paragraphs that result in a new map or topic document, specifies the name of an `<output>` element that defines the public and system identifier for the result document's document type declaration.
`@mapType`	Specifies root element type name for the map to be generated from the paragraph, e.g. "map".
`@prologType`	Specifies the tag name for the prolog for a generated map or topic. Defaults to "prolog" for topics and "topicmeta" for maps.
`@chunk`	For paragraphs that map or generate topicrefs, the value to set on the `@chunk` attribute in the result, normally "to-content" (indicating that the result navigation branch should be result in a single output "chunk" (e.g., single HTML page).
`@topicDoc`	For paragraphs that generate new topics, when set to "yes", indicates that the topic should be output in its own document. When set to "yes", should also specify a value for `@format`.
`@topicrefType`	For paragraphs that generate new documents and specify "yes" for `@topicDoc`, indicates the element type to use for the reference to the generated topic in the result map. Defaults to "topicref".
`@initialSectionType`	For styles that map to topic, indicates the name of a specialization of topic/section to use to contain any paragraphs within the topic that occur before an explicIty-indicated section. If not specified, the topic will not have an initial section. Useful for topic types, such as reference and lcLearningContent, that require all topic body content to be within some form of section.
`@outputclass`	The outputclass= value to use on the element directly generated from the incoming style (that is, the element specified by the tagName= attribute).

@topicOutputclass	For styles that generate topics, indicates the `@outputclass` value to use for the generated topic.
@sectionType	For styles that imply the creation of a containing section element, the element type to use.
@useAsTitle	For styles that map to a specialization of topic/section, indicates whether or not the paragraph content should be used as the section title. The default is "yes". If set to "yes", or unspecified, must set a value for `@tagName`.
@containerOutputclass	For styles that imply the creation of a container element (e.g., list items), specifies the `@outputclass` value for the container.
@navtitleType	For paragraphs that result in a `<topicref>` element, specifies the tag name to use for the topicref's navigation title element.

Common Style-to-Tag Mapping Cases

Describes how to map styles to tags for common cases

This section describes how to set up the mapping for different common cases.

But first some general tips:

- For paragraphs that map to topic components you must specify the `@topicZone` attribute as this attribute is used to group incoming paragraphs for further processing. If the `@topicZone` attribute is not specified a default of "body" is used. This is usually what you want but would be wrong if the paragraph should have gone in the map or topic prolog.
- Paragraphs that contribute to topic or map metadata can go anywhere in the Word document because they go into a separate topic zone and therefore get plucked out of the input and put in the right place regardless of where they occur.
- The `@level` attribute is what determines the hierarchical relationship among elements within the same output context, such as within the topic body.
- You can specify either the style display name, using `@styleName` or the style ID, using `@styleId`. The style name is the display name that is used in the various Word style-related user interfaces. The style ID is the value specified in the underlying markup for paragraphs and character ranges. If you specify both style name and style ID on a mapping, the style name takes precedence. In general, the style name is more reliable as Word may change the style ID, for example, when using locale-specific versions of Word.
- The "style ID" for Word styles is not the same as the display name of the style, but it is usually very similar. In most cases the style ID is the display name with all spaces and special character removed, e.g., the style with the display name "Heading 1" has the style ID "Heading1". If you happen to define two style names that differ only in their use of spaces, e.g., "Heading 1" and "Heading1", then the style ID of the second style defined will be something like "Heading10" so that the style ID is unique within the Word document. If you're not sure what the style ID is you can open the document.xml file from the DOCX Zip package and look for `<w:pStyle>` elements —the value of the `@w:val` attribute is the style ID.
- The `@tagName` attribute always specifies the name of the *primary* (most deeply nested) result element. The primary result may be surrounded by other elements as specified with other attributes, such as `@containerType`. The element named by `@tagName` is the element that will contain the text of the paragraph or character run being mapped, except in a few special cases, such as mapping to section elements.

Finally, the design of the current style-to-tag mapping document evolved somewhat organically based on client requirements as they came up. There are aspects of it that are not necessarily logically consistent. I have started the process of designing a new style-to-tag mapping document type but it's on the back burner. But I would definitely welcome any suggestions for how to make the mapping markup clearer or easier to use.

Simple Paragraphs and Character Runs

For paragraphs that simply map directly to elements within the topic body with no required container elements the general mapping specification is:

```
<style
  styleName="Normal"
  tagName="p"
  topicZone="body"
  level="1"
/>
```

Where the parts in italics are what you would change to match a specific Word paragraph style to a specific DITA output element.

This example maps a paragraph with the name "Normal" to the DITA element `<p>`.

Likewise, for character runs, you specify the style name and tag name:

```
<style
  styleName="Heading 1 Char"
  tagName="ph"
  topicZone="body"
  level="1"
/>
```

Note that as of version 0.9.16 there is no support for nesting character styles in the output, so if you need something like "bold italics" which you would typically markup like `<b><i>` in DITA you will need to define a single phrase-level element like `<bi>`. Note that the DITA for Publishers formatting domain includes elements for common combinations of bold, italic, and smallcaps, as well as other elements you might need. You can also use the transform's "final fixup" mode to convert single elements like `<bi>` into nested elements if you want.

Heading Paragraphs That Map to Topics

To map heading paragraphs to topics you specify the topic type and the markup details for the generated topic. The heading paragraph becomes the topic title and the other topic elements are generated automatically.

For example, to map the paragraph "Heading 1" to a concept topic you would use this mapping:

```
<style styleName="Heading 1"
  structureType="topicTitle"
  tagName="title"
  topicType="concept"
  bodyType="conbody"
  level="1"
  topicDoc="yes"
  format="concept"
  topicrefType="chapter"
/>
```

The structure type "topicTitle" indicates that this paragraph acts as a topic title, which means it will generate a new topic, either in a separate document or as a subtopic of a parent topic. In this case the value of the `@level` attribute ("1") indicates that this is a top-level topic.

The `@tagName` attribute defines the tagname to use for the topic's title element, "title" in this case (it would only be different from "title" if you were generating a specialized topic with a specialized topic title element).

The `@topicType` attribute specifies the tagname for the root element of the topic to be generated, "concept" in this case.

The `@bodyType` attribute specifies the tagname for the topic body element, "conbody" in this case.

The `@level` attribute indicates that this is a top-level topic and it would be subordinate only to a paragraph that specifies level zero, which is normally reserved for the paragraph that generates the root map or topic and should normally be the first unskipped paragraph in the Word document.

The `@topicDoc` attribute indicates that this topic should be put in a new document, which in turn implies the generation of a topicref to the generated document. The default is "no" so you must specify `@topicDoc` with a value of "yes" if you want the topic chunked out to a new file.

If you specify "yes" for `@topicDoc` then you must specify `@format`, which names an `<output>` element defined in the style-to-tag mapping document. The format value determines the public and system IDs to use for the generated topic document. You must also specify the `@topicrefType` attribute if process is generating a map in addition to topics.

The `@topicrefType` attribute specifies the tagname to use for the topicref that will refer to the generated topic. In this case the topicref tagname is "chapter".

Mapping lower-level headings uses the same pattern, specifying the appropriate level, e.g., level "2" for paragraph style Heading 2 and so on.

Mapping List Paragraphs (Generating Container Elements)

Lists are typical of elements that must be generated within the context of container elements, e.g. `<li>` within `<ol>` or `<ul>`.

To map list item paragraphs to DITA lists you map the paragraph to the appropriate list item element, e.g., `<li>` and then specify the name of the container element to wrap all adjacent list items in, specified using the `@containerType` attribute, like so:

```
<style
  styleName="List Bullet"
  tagName="li"
  containerType="ul"
  level="1"
  topicZone="body"
/>
```

Here the tag name is "li", indicating that the paragraph content will be output in a `<li>` element. The `@containerType` attribute names the tagname of the container element for the list item, `<ul>` in this case. The value "1" for the `@level` attribute means that this is the first level of thing within the topic zone (body). Any elements to be nested inside the list item would need to specify level "2" (for example, paragraphs for 2nd-level list items).

The implication of `@containerType` is that all adjacent Word paragraphs with the same container type will be output into a single instance of the specified container. This is how you can get a sequence of ListBullet paragraphs to output as a single DITA `<ul>` element.

This pattern of `@tagName` and `@containerType` can be used for any paragraphs that need to be output inside a common container that are not creating DITA `<section>` elements (which have special mapping support).

Mapping Definition Lists

Definition lists are a challenge because there is no single Word structure that maps directly to definition lists and DITA requires two-levels of containment: one for the definition list as a whole and one for each term/definition pair. To model definition lists you must use pairs of paragraphs: one for the definition term and one for the definition description.

Definition term paragraphs use the `@structureType` value "dt" and definition description paragraphs use the `@structureType` value "dd". For a pair of paragraph styles that represent a term/definition pair, they both specify the same value for the `@dlEntryType` attribute, e,g., "dlentry". Finally, they both specify the same value for the `@containerType` attribute, which specifies the overall definition list element, e.g. "dl".

Thus, given two paragraph styles "Def Term" and "Def Desc" you would define these mappings to generate a normal DITA definition list:

```
<style
  styleName="Def Term"
  structureType="dt"
  tagName="dt"
  dlEntryType="dlentry"
  containerType="dl"
  topicZone="body"
/>
<style
  styleName="Def Desc"
  structureType="dd"
  tagName="dd"
  dlEntryType="dlentry"
  containerType="dl"
  topicZone="body"
/>
```

Note that the resulting markup doesn't have to be a specialization of `<dl>` it just has to have the same structural pattern of two levels of containment with the lowest-level elements in common containers. You could use this mapping pattern to generate simple tables, for example.

Mapping Procedure Steps

Steps in procedures are similar to definition lists in that there are two layers of markup that wrap the paragraphs for a step: the `<steps>` element that contains each `<step>` and the `<cmd>` element that is the required first subelement of `<step>`.

Given a paragraph that is the first or only paragraph of a step, you can map it by treating it like a definition list, using a `@structureType` of "dt":

```
<style styleName="List 1"
  structureType="dt"
  dlEntryType="step"
  containerType="steps"
  tagName="cmd"
  level="1"
/>
```

Here the paragraph style "List1" is mapped to `<cmd>` within `<step>` within `<steps>`.

If you have follow-on paragraphs for the step they should go in an `<info>` element. For this case you would use normal mapping and specify level 2:

```
<style
  styleName="BodyText Step"
  containerType="info"
  tagName="p"
  level="2"
  structureType="block"
  topicZone="body"
/>
```

If you have list items that need to go in the `<info>` element then you use a definition list mapping like so:

```
<style
  styleName="Bullet 2 Step"
  containerType="info"
  dlEntryType="ul"
  tagName="li"
```

```
  level="2"
  structureType="dt"
  topicZone="body"
/>
```

Because the container type for both BodyText Step and Bullet 2 Step is "info" in this example both paragraph types will end up contributing to the same info element in the output if they occur together.

Mapping Tables

Word tables are automatically converted to DITA tables. Paragraph and character styles within table cells will be mapped as defined in the mapping but you don't have direct control over how the table elements map to DITA markup (but you can always post-process the initial DITA into whatever you want).

Skipping Paragraphs

It's often useful or necessary to have paragraphs in the Word document that shouldn't be reflected in the DITA output. For this case you can use a `@structureType` of "skip". Paragraphs with a structure type of "skip" are ignored and have no effect on output. In particular, they do not affect the determination of what paragraphs are adjacent for `@containerType` processing.

In addition to explicitly-skipped paragraphs, paragraphs that are not otherwise mapped and that have empty content (that is, they normalize to a single blank or to the empty string) are automatically skipped. This saves you having worry about users creating empty paragraphs to get vertical spacing in Word documents.

Mapping to Sections

DITA sections are challenging because they have three ways to represent the title: a literal title child element, an explicit `@spectitle` attribute, or a `@spectitle` set in the document type and not intended to be set by authors. In addition, some topic types require the use of sections within the topic body. This all leads to a bit of complexity.

The simplest case is where a paragraph acts as the title of section and should result in a section where the paragraph is section title and the subsequent paragraphs are within the section. For this you specify a `@structureType` of section and a `@tagName` of "title" (or whatever the section title element should be, but usually "title"):

```
<style
  styleName="My Section Title"
  structureType="section"
  tagName="title"
  sectionType="section"
  topicZone="body"
  useAsTitle="yes"
/>
```

Note that you don't have to worry about the `@level` attribute with sections because DITA sections cannot nest and must be direct children of the topic body. So the conversion processing will always do the right thing for sections.

Note, however, that if you have any sections within your topic body then all the paragraphs that follow the first section-creating paragraph will be in a section because there's no way to indicate that a given paragraph should not be in a section. However, this shouldn't be a problem in practice because it would be a very rare markup design that expected there to be a random mix of section elements and non-section elements within the topic body.

Any paragraphs that precede the first section-creating paragraph within a topic body will be direct children of the topic body *unless* you specify the `@initialSectionType` attribute for the paragraph that generates the containing topic.

For example, the Learning and Training learningContent topic type requires the use of section-type elements within the topic body. To handle this case you can specify the `@initialSectionType` attribute to indicate that the initial paragraphs of the topic body should be wrapped in a section.

For example, given a paragraph style "Lesson Title" that maps to a learningContent topic, you would define a mapping like this:

```
<style
  styleName="Lesson Title"
  structureType="topicTitle"
  initialSectionType="lcIntro"
  topicType="learningContent"
  bodyType="learningContentbody"
  level="1"
/>
```

This will result in markup like this:

```
<learningContent id="topicid">
  <title>Learning Content</title>
  <shortdesc>Put a shortdesc of one or two sentences here.</shortdesc>
  <learningContentbody>
    <lcIntro>
       ... (paragraphs go here)
    </lcIntro>
  </learningContentbody>
</learningContent>
```

All paragraphs up to the first paragraph that maps to section would go within the `<lcIntro>` element in this example.

In some cases paragraphs should map to a section, provide the title or spectitle, and then be the first paragraph of the section. For example, a paragraph like this:

```
Usage: Controls the construction of the section title.
```

Could provide the spec title of "Usage" and the initial paragraph of "Controls the construction of the section title.".

You can do this by indicating that the paragraph is not the section title and that the spectitle is the text up to the first colon:

```
<style styleName="My Section"
  structureType="section"
  tagName="p"
  topicZone="body"
  spectitle="#toColon"

/>
```

Here the keyword value "#toColon" for the `@spectitle` attribute indicates that the spectitle value should be taken from the paragraph content up to, but not including, the first colon. (Other values could be implemented but as of version 0.9.6 the only implemented value is "#toColon".) The value "p" for the `@tagName` attribute indicates that the paragraph will be the first paragraph of the section rather than the title.

You can also specify a literal value for `@spectitle`, which simply becomes the value of the `@spectitle` attribute in the generated DITA XML.

Mapping to Maps and Map Components (Topicrefs)

The simplest mapping is one in which a single Word document maps to a single result topic document. However, you can generate systems of maps in addition to topics from a single input Word file. However, this gets a little complex because there's a lot going on.

When you are generating a map and topics you must map the first non-skipped paragraph in the Word document to a map element, which will generate the root map of the output structure. It can also map to a topic (and by implication, a topicref to that topic) but it need not.

For example, if your first paragraph has the style "Publication Title" and you want to generate a bookmap map but not a topic you would use this mapping:

```
<style styleName="Title"
  level="0"
  structureType="mapTitle"
  tagName="title"
  format="bookmap"
  mapType="bookmap"
  prologType="topicmeta"
/>
```

The `@structureType` value of "mapTitle" triggers the general map generation. The other attributes define the details of the map markup: `@mapType` and `@prologType`. The value "0" (zero) for the `@level` attribute indicates that this is the root output structure. You should only have one level-zero paragraph-to-map or paragraph-to-topic mapping.

If you map a paragraph to a structure type of "mapTitle" at a level other than zero then you will create a submap that is referenced from the parent map (that is, the map that is one level up in the mapping hierarchy). In this case you must specify the `@topicrefType` attribute in order to get a topicref generated in the parent map.

When a paragraph maps to a map and a topic you specify a structure type of "mapTitle" and specify the topic type using the `@sectionStructure` attribute. You can then specify the other attributes used to generate a result topic as you would for a structure type of "mapTitle":

```
<style
  styleName="Topic Title"
  bodyType="learningContentbody"
  chunk="to-content"
  format="learningMap"
  initialSectionType="section"
  level="1"
  mapType="map"
  prologType="prolog"
  secondStructureType="topicTitle"
  structureType="mapTitle"
  tagName="title"
  topicDoc="yes"
  topicType="learningContent"
  topicrefType="learningContentRef"/>
```

This example maps a paragraph named "Topic Title" to a generic map (`@mapType` of "map") and a learningContent topic referenced from the map. The value of "1" for the `@level` attribute means this map will be a submap to the root map.

When a new document document should be referenced from the current map simply specify the `@topicrefType` attribute.

Mapping Metadata

TBD

Troubleshooting the Word-to-DITA Process

Tips on figuring out why the Word-to-DITA process isn't working correctly.

There are several ways in which the transform can go wrong:

- The markup generated is not valid against the DTD or schema the document is using
- The generated maps do not match the generated topics (for example, you get a topicref to a file that wasn't generated or a failure to generate a reference to a topic that was generated).

Failures can be caused by any of the following:

- Incorrect style-to-tag mapping specifications
- Incorrect styling in the Word document
- Bugs in the Word-to-DITA transformation code

If you have eliminated the first two causes then the issue is likely a bug. Do not hesitate to report bugs or suspected bugs. If this documentation didn't give you the information you needed to clearly distinguish user error from code bugs then that's a bug too and needs to be reported.

Your main debugging tools are:

- The Word-to-DITA transformation log.

  If you run the transform through the Toolkit then the Word-to-DITA transformation messages should be in the larger Toolkit log. They will be at the end of the log since the transform is the last thing run.

  If you run the transform directly then the log will be wherever XSLT messages go, which is a function of how you run the transform. I almost always run it from within Oxygen, which captures the messages in a separate window.

  If you are using the command line in Windows, be sure to set a large window buffer size in the propery settings for your command window. By default, you may only see the last 100 lines or so, which is usually not enough.
- Validation of the generated XML, either with the Tookit or in an XML-aware editor that is configured with the doctypes you're converting to.

  When invalid XML is generated it's usually pretty obvious what the issue is because it's a direct result of an error in the style-to-tag mapping specification and a validating editor should take you right to the point of error.
- Draft mode with the style names exposed in Word.

  In Word's draft mode view you can expose the style names along one edge of the editing pane. The exact way to turn it on differs from version to version. In Word 2007 for Mac it is the "style area width" setting within the View settings. Being able to see the style names makes it easier to spot places where the Word is not styled correctly or the mapping didn't account for something it should have.

General Troubleshooting Tips

The first tip is to isolate as many variables as possible.

This usually means creating the smallest test case that creates the problem. That makes it easier to work out what the problem is and provides a more convenient data set if you need to get outside help figuring out what the issue is.

Verify that the files you think are involved really are. Usually the easiest way to do this is to introduce syntax errors into files and verify that stuff breaks. If it doesn't break then you know something's not hooked up right.

Get a second pair of eyes. Often the most efficient way to spot an error is to explain the failure to someone else—you'll often realize the problem in the course of explaining it. If not, a fresh set of eyes will often spot something silly you've overlooked because you're too close to it.

Double check the style IDs. Style IDs are case sensitive and, as explained elsewhere, may not always match the display name for the style. A simple typo can throw things off and be hard to catch, especially if you've spent too long staring at your mapping specification. If you can, switch to using style names (new in version 0.9.16), which are not case sensitive and are easier to validate since the match what you see in Word itself.

The Markup is Not Valid Against the DTD

The first thing to check is that the map or topic is getting generated with the correct DOCTYPE declaration. This is determined by the value of the `@format` attribute on the `<style>` element that generates the map or topic and the corresponding `<output>` element that specifies the public and system identifiers to use in the DOCTYPE declaration.

If that is correct, then you have to examine the generated XML and see what didn't get generated correctly. Typical errors result from not generating container elements correctly, such as for list items.

If the mapping looks correct then the problem must be in the Word data. Verify that the Word document is styled correctly.

Also verify that your assumption about what the markup should be is correct. Create the markup you want to generate in an editor and verify that your understanding of what it should be is correct.

Generated Maps and Topics Do Not Match

As of version 0.9.10, there is an issue with paragraphs between the root title and the Word table of contents (that is, before the first generated topic) throwing off the map generation process so you get a reference to topic 1 but no topic 1 is generated.

The fix is to remove those paragraphs from the Word document or map them to skip or to metadata of the root map.

Word-to-DITA Style-to-Tag Mapping Video Tutorial

A video tutorial on setting up a non-trivial style-to-tag mapping for a typical technical document.

This video tutorials works through the start-to-finish process of creating a new style-to-tag mapping for a Word document that represents a typical technical document that becomes a tree of topics with concepts and tasks.

Note: This video predates release 0.9.16 and so shows using `@styleId` rather than `@styleName`. You should prefer `@styleName` rather than `@styleId`.

Note:

The video is served from the Screencast.com site.

External video link: *http://www.screencast.com/t/5NQQRhO1mXC*

Figure 1: Word-to-DITA Video Tutorial: Technical Document Style to Tag Mapping

Extending and Overriding the Word to DITA Transform

How to customize the XSLT processing to handle special cases

Any transformation driven by a declarative mapping will be limited in what can be done purely through the mapping. Thus the Word-to-DITA process is designed to be extended and customized to handle special cases.

The Word-to-DITA transformation is a three-phase transformation:

- Phase one processes the input DOCX document.xml file (and other files as needed) to generate an intermediate "simple word processing" document. The process is not intended to be extended. The "simple" document reflects the original paragraph and character run data from the Word document annotated with the details from the style-to-tag mapping.
- Phase two processes the intermediate "simple" document to generate the initial result DITA XML.
- Phase three is the "final fixup" phase, which processes the inital result DITA XML to generate the final DITA XML. By default the final-fixup is just a passthrough stage but can be extended to do cleanup as needed.

In addition, the transformation provides default rules for generating output filenames and for generating element IDs, both of which can be overridden.

You extend and override the base transformation by creating a top-level XSLT document that imports the docx2dita.xsl transformation from the word2dita Toolkit plugin and implements any override or extension templates required. You can deploy this override in a separate Open Toolkit plugin that depends on the base word2dita plugin and that defines its own transformation type or you can specify the w2d.word2dita.xslt Ant parameter with the full path and filename of your custom transformation.

The second and third processing phases are implemented by the `simple2dita.xsl` file.

The extension points provided are:

- The "final fixup" phase, which you can extend by implementing templates in the "final-fixup" mode. These templates operate on the DITA XML produced by the the second phase and must produce valid DITA markup as their output. Note that the XML handled by the final-fixup mode doesn't have any associated schema or DTD at that point, so there are no `@class` attributes to key on. This means you must use templates that match on the element type names, rather than on `@class` attribute values.

- The "generate-id" and "topic-name" modes can be overridden to provide custom ID generation logic. The mappings for individual paragraph and character styles can specify a named "ID generator" value that is then passed as a parameter ot the generate-id mode. Custom ID generator code can use that parameter to select the appropriate ID generation logic.
- The "topic-url"mode can be overridden to implement custom rules for constructing result topic filenames.
- The "map-url" mode can be overridden to implement custom rules for construction result map filenames.

Sample Custom Word-to-DITA XSLT Stylesheet

A working example of a custom Word-to-DITA XSLT stylesheet

To customize and extend the base Word-to-DITA transformation you need a new top-level XSLT document that includes `docx2dita.xsl` transformation and then adds any new templates you need in order to customize processing. The following example shows an example of extending the "final-fixup" mode to capture literal numbers in topic titles.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs "
  version="2.0">

  <xsl:import href="../../net.sourceforge.dita4publishers.word2dita/xsl/
docx2dita.xsl"/>

  <xsl:template match="concept/title" mode="final-fixup">
    <!-- Look for numbers for the form "1-1" at the start of titles and
wrap in d4pSimpleEnumeration
         element. The number must be in the first text node, not in a
subelement.
    -->
    <xsl:variable name="childNodes" select="./node()" as="node()*"/>
    <xsl:message> + [DEBUG] childNodes=<xsl:sequence select="$childNodes"/
></xsl:message>
    <xsl:choose>
      <xsl:when test="$childNodes[1]/self::text() and
matches($childNodes[1], '^[0-9]+')">
        <xsl:copy>
          <xsl:apply-templates select="@*" mode="#current"/>
          <xsl:analyze-string select="$childNodes[1]" regex="^(([0-9]+(-
[0-9]+)*)[ ]+)(.*)">
            <xsl:matching-substring>
              <d4pSimpleEnumeration><xsl:sequence select="regex-group(1)"/
></d4pSimpleEnumeration>
              <xsl:sequence select="regex-group(4)"/>
            </xsl:matching-substring>
            <xsl:non-matching-substring>
              <xsl:sequence select="."/>
            </xsl:non-matching-substring>
          </xsl:analyze-string>
          <xsl:apply-templates select="$childNodes[position() > 1]"
mode="#current"/>
        </xsl:copy>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-imports/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

This transform as shown is intended to be packaged in a separate Open Toolkit plugin so that it is in a known location relative to the base transform provided by the Word-to-DITA. In this example the transform is in the directory `xsl/` under the plugin's main directory (mirroring the organization of the word-to-DITA plugin itself).

The plugin descriptor looks like this:

```
<!--
  Plugin descriptor for an example Word-to-DITA extension transform

  Use this as a sample for your own plugin.

  -->
<plugin id="org.example.d4p.word2ditaextension">
  <require plugin="net.sourceforge.dita4publishers.word2dita"/>

  <!-- This plugin just provides the transform in a reliable location
       relative to the base transform. It doesn't define its own
       transformation type.
    -->
</plugin>
```

The directory structure of the plugin is:

```
org.example.d4p.word2ditaextension/
  plugin.xml
  xsl/
    sample-word-to-dita-customization.xsl
```

You would copy the `org.example.d4p.word2ditaextension` directory to the `plugins` directory of your Toolkit in order to make it available. Note that because this plugin doesn't define a new transformation type or directly-extend any other plugins, you don't have to run the integrator.xml script after deploying it.

To use the customization you would specify the XSLT file as the value of the w2d.word2dita.xslt Ant parameter, either on the command line or in an Ant build script that applies the Word-to-DITA process to a specific file.

A working version of this plugin is included in the DITA for Publishers Open Toolkit plugin package as the plugin "org.example.d4p.word2ditaextension".

Tips for Using Word With the Word-to-DITA Transform

Things you can or should do to make the transformation most effective

All inline content that should generate DITA markup must be styled with a character style. This avoids complications around interpreting arbitrary formatting into the appropriate markup. This may require defining new character styles for bold, italic, and so on.

You can use Word's search and replace feature to find arbitrary styling and replace it with the use of specific character styles.

Likewise, you can use Word's search and replace to change one style name into another. This makes it relatively easy to migrate existing Word documents to new style names if you change the style names for some reason.

If you are defining new styles in order to facilitate mapping to DITA it is a good idea to name the styles so that all the styles used in a particular structure start with the same prefix. This will cause the styles to be grouped together within the Word style list. Otherwise users have to jump around in the style list to find the right style, which is very annoying.

If you are setting up styles to facilitate direct authoring (rather than after-the-fact style application) then be sure to set up the "next style" setting for each style wherever possible.

If authors will be creating deeply-nested structures use outline levels with paragraphs so that the Word outline view reflects the intended DITA structure as much as possible.

Encourage authors to use Draft mode and turn on the paragraph style display—this makes it much easier for authors to see what is really going on the Word document and helps avoid styling errors.

Use the Word style and template manager to keep the set of styles in your documents and templates as focused as possible.

As much as possible prefer a "manuscript" visual style over a WYSIWYG style so that it's clearer to authors that what they create visually in Word will not necessarily translate directly to the XML.

Word-to-DITA XSLT Transformation Parameters

Required and optional parameters for the Word-to-DITA XSLT transformation

To run the docx2dita.xsl transformation outside the Toolkit use the XSLT parameters shown here.

Required Parameters

outputDir

The directory to put generated output in. Should be an absolute URL.

styleMapUri

The URI of the style-to-tag map to use. Should be specified as an absolute URL, e.g., "file:// workspace/word2dita/xsl/mystyle-to-tag-map.xml".

Optional Parameters

fileNamePrefix

The prefix to use for generated filenames other than subordinate maps. Default is "" (no prefix).

filterBr

When set to "true" filters out all literal break characters. If not specified, all literal breaks in the word document are preserved in the intermediate transform and, by default, in the resulting DITA XML (using the DITA4Publishers `<br>` element from the formatting domain).

filterTabs

When set to "true" filters out all literal tab characters. If not specified, all literal tabs in the word document are preserved in the intermediate transform and, by default, in the resulting DITA XML (using the DITA4Publishers `<tab>` element from the formatting domain).

includeWordBackPointers

When set to "true", the resulting DITA files include `@xtrc` attributes that contain XPath expressions that locate the Word paragraph element from which the DITA element was generated. Set to "false" to turn this off. Default is "true" (`@xtrc` attributes are generated).

language

Specifies the value to use for the `@xml:lang` attribute, e.g. "ja-JP". If not specified, the value "en-US" is used.

mediaDirUri

The URI of the folder to put graphics in (determines the value of `@src` and `@href` attributes for generated graphic references). Default is "topics/media" under the main output directory.

rawPlatformString

The operating system platform as provided by the Ant os.name property. Default is "unknown".

rootMapName

The name (without extension) to use for the root map. Default is "rootmap".

rootMapUrl

The full filename the root map. Constructed from the rootMapName parameter by default.

rootTopicName

The name (without extension) to use for the root topic. Default is the base name of the input DOCX file.

submapNamePrefix

The prefix to use for the filenames of generated submaps. Default is "map".

topicExtension

The filename extension to use for topics. Default is ".dita".

Word-to-DITA Ant Parameters

Ant parameters for the Word-to-DITA transformation type

To run the Word-to-DITA transformation using the Open Toolkit plugin, specify the following Ant parameters.

Required Parameters

args.input

The absolute filename and path of the DOCX file to process.

word.doc

The path and filename of the Word DOCX file to process.

Optional Parameters

output.dir

Specifies the absolute path of of the directory to contain the generated output.

w2d.clean.output.dir

Set to "true" to have the output directory cleaned out before running the transform. The default is "false".

w2d.debug.xslt

Turns on debugging of the XSLT transform. Default is "false".

w2d.defaultStyleToTagMap

Specifies the location of the style-to-tag map to use as the default if no style-to-tag-map is specified. The default is the word-builtin-styles-style2tagmap.xml file. This parameter is mostly useful for setting as a default in installation-specific properties files. You would not normally specify it for a specific document.

w2d.filename.prefix

The prefix to use for generated files other than submaps. The default is no prefix.

w2d.filter.br

Control whether or not `<br>` elements generated from the Word data will be filtered out of the result DITA. Set to "true" to filter out `<br>` elements. The default is "false" (`<br>` elements are included in the result DITA). The `<br>` element is provided by the DITA for Publishers formatting vocabulary domain module.

w2d.filter.tabs

Control whether or not `<tab>` elements generated from the Word data will be filtered out of the result DITA. Set to "true" to filter out `<tab>` elements. The default is "false" (`<tab>` elements are included in the result DITA). The `<tab>` element is provided by the DITA for Publishers formatting vocabulary domain module.

w2d.include.word.backpointers

Set to "false" to turn off inclusion of `@xtrc` attributes in the generated DITA files that point back to the original paragraphs in the Word document. Default is "true" (`@xtrc` attributes are generated).

w2d.root.map.name

The filename (without extension) to use for the root map if one is generated. By default uses the filename of the input DOCX file. Only specify this property if the style-to-tag mapping will result in a map. If the style-to-tag mapping only produces topics, then do not specify this property.

w2d.root.output.filename

Specifies the complete filename (including extension), of the root output file, whether it's a topic or a map. You would not normally specify this property unless the behavior of w2d.root.map.name or w2d.root.topic.name doesn't give you the result you want.

w2d.root.topic.name

The filename (without extension) to use for the root topic. By default, uses the filename of the input DOCX file. This property is ignored if w2d.root.topic.name is also specified.

w2d.style-to-tag-map

The absolute filename and path of the style-to-tag mapping to use. If not specified, the built-in mapping file `net.sourceforge.dita4publishers.word2dita/xsl/word-builtin-styles-style2tagmap.xml` is used.

w2d.submap.name.prefix

The prefix to use for generated submaps. The default is "map".

w2d.temp.dir

The temporary directory to use for word2dita process. Default is ${basedir}/temp.

w2d.topic.extension

The extension to use for generated topic files. The default is ".dita".

w2d.word2dita.xslt

Specifies the name of the XSLT transform to use to convert the DOCX file to DITA. Default is the `docx2dita.xsl` transform in the word2dita Toolkit plugin.

The DITA-to-InDesign Transformation Framework

The DITA-to-InDesign transformation framework enables the generation of InDesign documents and InCopy articles from DITA maps and topics.

The framework consists of both a library of XSLT transforms for generating InCopy articles and a Java library for reading and writing InDesign interchange (INX) files. Generating InDesign from DITA content almost always requires some degree of customization and may require some Java programming if you need to generate complete InDesign documents.

The DITA-ot-InDesign framework is intended to provide a low-cost, easy-to-configure path from DITA content to InDesign for publishing PDF. It has been developed primarily to meet the needs of typical Publishing business processes, such as producing magazines highly-design books. It does not and cannot provide 100% automation of print layout except in very narrow cases. Because the process is a one-way process that is not directly connected to InDesign itself it is inherently limited in the amount of layout automation it can provide. But it is free.

Where full automation through InDesign is required the Typefi product provides up to 100% automation for publications where the number of titles that use a particular layout design is large enough to justify the investment in setup. See *www.typefi.com* for more information on the Typefi product.

The DITA-to-InDesign framework is *not* intended and does not attempt to provide round tripping between XML and InDesign. While it is technically possible to do some round tripping when certain constraints are imposed, the presumption of the DITA-to-InDesign process is that, because the InDesign can be regenerated on demand, the need to make changes in InDesign should generally be minimized. Likewise, the cost and complexity of a round-trip-based system is much greater than the cost of managing last-minute text changes can generally justify. The DITA for Publishers formatting domain vocabulary module does provide markup that could be used to capture formatting details but the DITA-to-InDesign tools to not take advantage of this markup in any way.

In the narrower case of InCopy articles (just the text content), it is possible to implement reliable round tripping between XML and InCopy using relatively simple XSLT transforms. The InCopy and INX knowledge represented in the DITA-to-InDesign XSLT libraries can be used for this purpose but the project does not provide any examples of doing so.

Finally, the DITA-to-InDesign process does *not* use InDesign's built-in XML handling facilities in any way. These facilities are too limited to allow a general and fully-functional solution to generating best-possible InDesign documents from XML. Rather, the DITA-to-InDesign framework is predicated on generating complete InDesign documents directly, which provides full control over all aspects of the final InDesign document except final layout, which can only be done by InDesign itself.

Note: As of release 0.9.11 of the DITA for Publishers materials there is no general DITA-optimized InDesign template that would enable generic DITA-to-InDesign transforms. Providing such a general-purpose template has always been a design goal of the DITA-to-InDesign project but to date the users of the DITA-to-InDesign technology have not needed it because they have all had pre-existing InDesign templates and paragraph style sets. Development of such a general-purpose template is on the short-term development plan for DITA-to-InDesign, such as it is.

The DITA-to-InDesign framework can be used in any one of the following ways:

1. Generate just InCopy articles (text content only) that are then manually placed into InDesign documents.

   This process is appropriate for highly-designed and variable publications such as magazines where it would be impossible or impractical or just not very helpful to generate the InDesign document as well. This is the simplest use of the DITA-to-InDesign framework as it can involve only the DITA-to-InCopy XSLT transform.
2. Generate InCopy articles and a corresponding InDesign document, based on a template InDesign document, that links in each InCopy article.

   This process is appropriate for documents where the page layout and general arrangement of pages and frames does not change from title to title, such as less-highly-designed magazines or books with consistent styles. This process can only provide limited layout automation, essentially limited to the first page or first two pages of each article. More automation is possible through custom InDesign scripting or when the individual articles have limited or invariant scope, such as always being exactly one page.

   The output of this process is a initial set of InCopy articles and an InDesign document ready for refinement by a Designer. Because the articles are linked from the InDesign document, they can be updated (regenerated or modified by hand) without disturbing the page layout defined in InDesign. This can allow authors and designers to work in parallel as the content is being developed and completed. The main limitation is that any formatting changes made to InCopy articles by a Designer are unavoidably lost if the article is regenerated.
3. Generate a single InDesign document with all stories included directly (not as separate InCopy articles).

   This process is appropriate when the layout is mostly or entirely automated or there is some other reason not to keep the articles separate. This approach generally presumes that the Designer will touch the InDesign document once and then be done with it.

Options (2) and (3) require use of the Java INX support library in order to generate the InDesign document. While INX is nominally an XML format, manipulating it is pretty much beyond what on can do productively with XSLT. The INX support library provides a full abstract data model for manipulating and creating INX documents, allowing you to do anything that doesn't require knowledge of how flowing content is formatted. For example, you can create new spreads, pages, and frames, create or link stories or media files, calculate the relative geometry of frames and so on.

Note: The Java INX processing is currently not usefully parameterized or configuration driven. Work is being done on this but to date use of the INX library has been very ad-hoc and client-specific.

Generating InCopy and InDesign effectively requires some understanding of how InDesign works and general techniques for optimizing your InDesign templates and paragraph and character styles to enable maximum automation. Many of these optimizations are not things that Designers normally do so you will likely need to work closely with your Designer to help them understand how to add the necessary enhancements to your InDesign templates.

This section uses the term "Designer" to mean anyone preforming the task of creation or modification of InDesign documents, and in particular, the design of the visual aspects of documents such as page design, typographic details, and so on, as well as doing the manual work required to create final-form printed documents uisng InDesign. One of the design goals of the DITA-to-InDesign tools is to automate the manual aspects of this job as much as possible, allowing Designers to focus on the creative design aspects of documents, rather than the non-creative tasks of simply getting content into InDesign.

DITA-to-InDesign does not change the roll of Designers as Designers, that is, as the implementors of specific visions for the visual presentation of information because it leaves the design task in the design tools domain rather than moving it into the data processing domain as in the case of batch formatting systems such as XSL-FO and similar non-interactive composition software.

Overview of the InDesign and InCopy Products

Adobe InDesign and InCopy are companion products, part of Adobe's Creative Suite set of products. InDesign is a desktop publishing tool widely used in Publishing and where typographic quality and layout sophistication are paramount.

Adobe InDesign is a general-purpose desktop publishing system primarily intended for the interactive creation of print publications with high typographic quality and arbitrarily complex page layouts. It is not an automatic composition tool, although it enables a degree of automation through scripting using Adobe ActiveScript (a form JavaScript). InDesign is also available as a server, the separately-licensed InDesign Server product, which enables batch processing of InDesign documents. InDesign documents are stored natively as binary files. They may also be exported and imported in two different XML formats: The InDesign Interchange format (INX) and the InDesign Markup Language (IDML). The DITA-to-InDesign framework works with the INX format exclusively.

Adobe InCopy is a companion product to InDesign intended for editorial authoring. It provides the "story editor" component of InDesign, allowing authors to create content without the ability to modify the visual layout or design of documents. InCopy documents ("articles") are stored natively as XML files (INCX). InCopy articles may be linked into InDesign documents or they may be copied into InDesign documents, as appropriate. InCopy articles that are linked can be updated independently from the InDesign documents that use them, whereby the InDesign document provides the option to use the updated content.

The division between InDesign for design and typography and InCopy for content provides a useful separation of concerns that the DITA-to-InDesign process takes advantage of. It allows Designers to focus on the design aspects of documents: the page layouts, paragraph and character style details, etc., while content authors can focus on the content without fear of disrupting the design. InCopy also provides some simple collaboration features by which multiple authors can work on the same article from a shared storage area such as a network drive.

An InDesign document consists of a sequence of "spreads", where a spread is one or more pages and some number of frames. Frames contain either text or graphics and thereby hold the content of the document. Pages have associated "page masters", which define the static components of the page (headers, footers, etc.) and can also provide template frames for use in individual page instances.

Frames may be "threaded" together to create a sequence of frames through which content will flow automatically when placed into the first frame of the sequence.

Text content is organized into "stories", where a story is a sequence of paragraphs. Stories are "placed" into documents by associating them with text frames. A given text frame can be associated with at most one story.

A story may be stored within the InDesign document or stored externally as an InCopy article and linked into the InDesign document. Linked stories can be updated outside of InDesign and InDesign will detect the new version and give the InDeisgn operator the option of updating InDesign's internal copy of the story.

InDesign documents can also link in graphics, which are placed into graphic frames. InDesign supports a wide variety of graphic formats.

InDesign documents can be rendered to PDF and EPUB among other formats.

Generating InDesign from DITA Using the Toolkit Plugin

How to use the DITA-to-InDesign Toolkit Plugin to generate inDesign and InCopy documents from DITA content.

To use the DITA-to-InDesign transform as a Toolkit plugin simply deploy the DITA for Publishers Toolkit plugins to your Open Toolkit directory. The DITA-to-InDesign transformation type is "indesign".

Required Parameters

None.

Optional Parameters

output.dir

Specifies the absolute path of the directory to contain the generated output.

title.only.topic.class.spec

Specifies the `@class` value to use for topics generated from topic heads.

title.only.topic.title.class.spec

Specifies the `@class` value to use for the titles of topics generated from topic heads.

Configuring the DITA-to-InDesign Transformation

You can generate InCopy articles from single topics or from maps using XSLT.

An InCopy article contains a single sequence of paragraphs to be placed within an InDesign frame sequence. The native format for InCopy articles is INCX, an XML format, which makes it possible to generate InCopy articles using XSLT.

The DITA-to-InDesign framework provides a general XSLT library and base transforms for generating InCopy articles from DITA source (although the library itself is not particularly DITA-specific, meaning it can be applied to sort of XML without too much effort).

The base transforms are packaged in the org.dita2indesign.dita2indesign Open Toolkit plugin. The dita2indesign transformation type provides the extendable XSLT transform dita2indesign.xsl. You can use normal Toolkit extension facilities to extend this transformation using extension point ID "xsl.transtype-indesign'.

You can also create a custom top-level XSLT transform that imports the transform `dita2indesignImpl.xsl` and either define a new transformation type or simply specify the transformation file as the value of the map2indesign.style Ant property.

The basic purpose of the transformation is to map XML elements to InDesign paragraphs and character runs with the appropriate paragraph and character styles applied. All of the visual aspects of the rendition, with the exception of explicit frame breaks, are provided by InDesign through the style definitions and other layout aspects, which are defined either manually by a Designer or generated through a separate, Java-based InDesign generation process.

Thus the primary configuration task is to map elements in context to style names. This is done through an XSLT module that uses very simple XSLT templates to do the mapping. You do not need to have any knowledge of XSLT details in order to define this mapping. However, you must know at least enough XPath to be able to specify the match expressions needed by your data. However, this is a pretty small subset of XPath in most cases. The examples provided in this section should be sufficient for most tasks.

Another configuration task that may be required is emitting frame, column, or page break characters as required by your content to trigger the appropriate flowing of text through sequences of threaded frames. This is done through XSLT templates that override the base processing for paragraphs. Again, no prior XSLT knowledge is required, you simply need to copy the examples in this section and adapt them as appropriate to your content.

For more involved requirements you may need to perform more complex processing, such as using metadata values to generate paragraphs in the output, reorder elements from the input to the output, or generate multiple InCopy articles from a single topic. This requires more general knowledge of XSLT but examples of these types of processing are available to use as guides and starting points.

Mapping XML Elements to InDesign Paragraph and Character Styles

You define the mapping of elements to specific character and paragraph styles using simple XSLT templates.

Unlike the Word-to-DITA transformation framework, which uses a separate configuration file to define the mapping of Word paragraphs to DITA elements, the DITA-to-InDesign process uses XSLT modules with very simple XSLT templates. The reason for this is that the only way to fully support mapping arbitrary elements in context to styles is to use XPath or something equivalent to it and it would be harder to define and implement a separate configuration specification that gave you that power than it would to simply code the XSLT directly.

The base DITA-to-InDesign transform includes a module named `elem2styleMapper.xsl`, which defines the default mapping for DITA elements to paragraph and character style names. However, it is unlikely that this mapping will work for some or even any of your content, since the style names need to match your InDesign templates. Thus you will likely need to create your own element-to-style mapper module that extends or overrides the base mapping, either as an extension to the DITA2InDesign plugin or via a custom top-level transform that includes your custom mapper as well as the base DITA2InDesign transform.

The element-to-style mapper looks like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:local="urn:local-functions"
      xmlns:df="http://dita2indesign.org/dita/functions"
      xmlns:e2s="http//dita2indesign.org/functions/element-to-style-mapping"
      exclude-result-prefixes="xs local df e2s"
      version="2.0">

  <!-- Element-to-style mapper

    This module provides the base implementation for
    the "style-map" modes, which map elements in context
    to InDesign style names (paragraph, character, frame,
    object, table).

    Copyright (c) 2009 Really Strategies, Inc.

    NOTE: This material is intended to be donated to the RSI-sponsored
    DITA2InDesign open-source project.
  -->
  <xsl:import href="../lib/dita-support-lib.xsl"/>
  <xsl:import href="lib/incx_generation_util.xsl"/>

  <xsl:template match="/*[df:class(., 'topic/topic')]/*[df:class(., 'topic/
title')]"
    mode="style-map-pstyle">
    <xsl:sequence select="'Topic Title 1'"/>
  </xsl:template>

  <xsl:template match="/*[df:class(., 'topic/topic')]/*[df:class(., 'topic/
topic')]/*[df:class(., 'topic/title')]"
    mode="style-map-pstyle">
    <xsl:sequence select="'Topic Title 2'"/>
  </xsl:template>

  ...

  <xsl:template match="*[df:class(., 'topic/ph')]"
    mode="style-map-cstyle" priority="0.75">
    <xsl:sequence select="'[No character style]'"/>
  </xsl:template>

  <xsl:template match="*[df:class(., 'topic/cite')]"
    mode="style-map-cstyle">
    <xsl:sequence select="'italic'"/>
  </xsl:template>

  ...
```

```
</xsl:stylesheet>
```

The module provides templates for two different XSLT modes; style-map-pstyle and style-map-cstyle.

The style-map-pstyle templates map elements in context to paragraph style names. The style-map-cstyle templates map elements to character style names. Note that the same XML element may map to both paragraphs and character runs in different contexts.

Each template does nothing other than produce a literal string that is the InDesign style name.

Looking at the first highlighted template above, the parts are:

Component	**Meaning**
`match="/*[df:class(., 'topic/topic')]/*[df:class(., 'topic/title')]"`	The match statement that matches a specific context. The value of the `@match` attribute is an XPath expression. Here it uses the DITA Support Library "df:class" function to match on elements based on their DITA `@class` values rather than their tagnames, e.g., title within a topic that is the root of the document (as indicated by the leading "/" in the XPath expression).
`mode="style-map-pstyle"`	The XSLT mode this template is used in, in this case "style-map-pstyle", indicating a template that maps to a paragraph style name. The other possible mode is "style-map-cstyle"
`<xsl:sequence select="'Topic Title 1'"/>`	An XSLT 2 sequence constructor used to produce the literal string "Topic Title 1" per the value of the `@select` attribute. Note the single quotes within the double quotes of the attribute specification. The style name string is the style display name as it appears in the InDesign user interface.

The second highlighted example follows exactly the same pattern, differing only in the name of the mode ("style-map-cstyle") .

Your custom element-to-style mappings should follow the same pattern, differing only in the details of the `@match` expression and the style name produced.

The only otther thing you may need to add is a `@priority` attribute when you need to have templates for more-specialized elements that would otherwise match on a template for the specialization's ancestor types. In that case, you must specify a `@priority` attribute on the `<xsl:template>` element with a value greater than 1, e.g., `<xsl:template priority="10" ...` or whatever number ensures the template will match before other templates the element might have matched on.

For example, say your vocabulary includes a specialization of `<ph>` named "foo" and a specialization of "foo" named "bar". You want `<foo>` to map to the paragraph style "Foo" but `<bar>` to map to the paragraph style "Bar". You start by creating these two match templates:

```
  <xsl:template match="*[df:class(., 'foodomain-d/foo')]"
    mode="style-map-pstyle">
    <xsl:sequence select="'Foo'"/>
  </xsl:template>

  <xsl:template match="*[df:class(., 'bardomain-d/bar')]"
    mode="style-map-pstyle">
    <xsl:sequence select="'Bar'"/>
  </xsl:template>
```

If you used these templates as is you would get "ambiguous rule match" messages for <bar> elements because <bar> is a specialization of <foo> and thus matches both on a check for foo as well as bar (that is, the @class value for <bar> would be something like "+ topic/ph foodomain-d/foo bardomain-d/bar ").

To remove the ambiguity you would add a @priority attribute to the template for <bar>, forcing it to match first:

```
<xsl:template match="*[df:class(., 'bardomain-d/bar')]"
  priority="10"
  mode="style-map-pstyle">
  <xsl:sequence select="'Bar'"/>
</xsl:template>
```

Now when you run the transform all the <bar> elements will map to the style 'Bar' as you intended.

The @priority value of "0.75" for the template for the <ph> element in the example above puts it's priority *below* the default priority of "1", ensuring that the template for <ph> will not interfere with any templates for specializations of <ph>.

To create your own mapping module, simply copy the base elem2styleMap.xsl file to a new location, normally a Toolkit plugin for your extensions, remove all the templates you don't want to override, modify the ones you do, and add any additional mappings you need.

If these mappings should be global to all InCopy generations in your environment you can use the module as an Toolkit-provided extension to the base DITA2InDesign transformation type.

If, as is more likely, your mappings are for a specific use and not global to all InCopy transforms, then you must create a new plugin with a new top-level transform that includes your mapping module and the base dita2indesign.xsl transform.

Preparing InDesign Templates for DITA-to-InDesign Use

By taking advantage of InDesign features you can maximize the automation provided by the DITA-to-InDesign framework.

The main optimization techniques are:

- Using threaded frames with frame break characters to automate layout of complex page designs such as chapter openers.
- Defining separate page masters for each distinct page layout (meaning unique arrangement of frames)
- Defining paragraph styles for each distinct structural component that needs a distinct presentation, especially the parts of lists (first, last, middle items, etc.).
- Using script labels on frames to enable automated placement and generation of frames based on rules applied to the XML content.

Thread Frames for Complex Page Layouts

The most important automation enabler is the use of threaded frames coupled with the use of "frame break" characters in the generated content.

For example, a chapter opener might use several frames to lay out the chapter number, chapter title, the opening paragraph, an epigraph or epigram, and so on. When Designers are creating such pages by hand they typically created disconnected (unthreaded) frames for each part and simply place the content in the frames manually.

However, the same layout can be automated by threading all the frames in the appropriate order and using Adobe-specific frame break characters in the input to force text to start in the appropriate frame. As long as the generated text doesn't overflow a frame before the frame break is reached, all the text will be layed out correctly.

You should always be able to take an existing page design with unthreaded frames and simply thread the frames together to create the appropriate page master.

The generation of frame break characters is defined in the XSLT transform that generates the InCopy articles.

Use Page Masters for All Distinct Page Layouts

Designers will often not bother to set up distinct page masters for different pages where the variations are minor or where it's easier for them to simply do it manually. However, for automation there needs to be a distinct page master for each distinct page design, where a page design is distinct when it has frames not found on any other page in the same position.

For example, odd and even versions of the same page need to have distinct page masters.

Having distinct page masters allows the InDesign generation process to both select the correct page master for a given XML context and also to select the correct frame from a page master when it needs to generate a frame on demand.

In general, the more distinction there is in the InDesign template, the easier it is to transform to it because it requires fewer complex business rules in the generation code.

Page master names are used by the Java-based InDesign generation process to choose the appropriate page master as the basis for new pages generated for XML content.

Define Styles for All Visual Distinctions

InDesign allows you to create named styles for paragraphs, character runs, objects, and tables. You can then use these styles to automate much of the layout of text and graphics generated from the XML.

Most designers do not bother to create distinct styles for paragraphs that require special treatment, such as the initial or last items in lists, because it's just as easy for them to do the manual adjustment as it is to apply a distinct style. But when generating the paragraphs all visual distinctions should be defined by styles, which requires distinct styles for different instances of the same basic element, lists being the most obvious example.

Likewise, you can use table and object styles to better automate the visual aspects of tables and graphics.

Style names are used in the XML-to-InDesign mapping.

General Style Definition Guidelines

In general, you want to define as much of the visual aspect of the paragraph as you can using the style definition alone. This includes:

- All spacing before, after, and around the paragraph.
- Any generated text for the paragraph, including list bullets, numbers, and so on.
- For numbered items, use InDesign's automatic numbering features for lists.

  Automatic numbering is usually not usable for numbers that span topic boundaries, such as heading and figure numbers. For those items the numbers should be generated as part of the InDesign generation process.
- Use nested styles to format generated text. You can use nested styles to format literal text in the InDesign paragraph but it's usually easier to just generate the appropriate character style as part of the XML-to-InDesign generation process.
- Derive variant styles from base styles to make style development and maintenance easier. For example, all the paragraph styles for a given list type should be based off the same base paragraph style.
- Name styles clearly and consistently.

  It seems to work best to give styles a common prefix with suffixes that indicate the purpose of variants, such as "Bullet List 1", "Bullet List 1 First", etc. Many designers use the convention of putting character style names in all lowercase. Because the style names will be specified in the XSLT code that generates the InDesign content, it's important to have consistent names that reduce the chance of error when defining or updating the mapping.

The visual aspects of the style definitions don't matter at all to the transformation process, only the style names. This means that different documents can use the same style names with different visual definitions to get very different looks from the same transformation mapping, simply by using different InDesign templates.

If you are using Microsoft Word to author documents from which XML is generated that then gets used to generate InDesign there is no general requirement to have the Word style names match the InDesign style names. However, it often makes things clearer to users and implementors if there is a clear association between Word style names and InDesign style names. In many cases pre-existing practice is to flow Word documents into InDesign, where the style

names match so that InDesign automatically applies appropriate styles. There is no reason to change this practice when putting XML between Word and InDesign.

By the same token, because you are going through a neutral intermediate format (XML) on the path from Word to InDesign, there is no technical requirement the Word styles have any association to the InDesign styles as the mappings from Word to XML and from XML to Word are completely independent.

Styles for Lists

For lists, you generally need these styles, one set for each type of list (bulleted, numbered) and indention level (first, second, third, etc.):

- A separate style for the first item in the list. This style will control the amount of space between the preceding paragraph and the start of the list.
- A separate style for the main (not first) list items. This style controls the space between items.
- A separate style for the last item in the list if you require different spacing after the list than following paragraphs would normally generate.
- Paragraphs for second and subsequent paragraphs within a list item.
- Paragraphs for other elements that might occur with list items and that would need special treatment, such as notes, examples, and so on.

To Designers this will seem like a very large set of styles but it's necessary in order to enable automation.

Use InDesign auto numbering for numbered lists unless there is some reason that the InDesign process must generate the numbers (for example, because they reflect arbitrary author control over numbering or a more complex DITA-defined business rule, such as task numbering).

Object Styles

InDesign lets you define styles for objects (frames) and styles. These styles can automate some aspects of graphics and table formatting.

Use object styles by defining object styles in your InDesign template document and using those styles on template frames used for graphics.

Tables styles are not directly supported by the general DITA-to-InDesign transformation framework as of version 0.9.11.

Use Script Labels for Key Frames

In this context "key frame" means a frame that is the initial frame of a text flow (sequence of threaded frames) or a "template" frame to use for holding graphics. That is, frames that need to be created or located by the InDesign generation process in order to construct new pages or properly populate existing pages.

InDesign provides a general feature, script labels, that lets Designers associate arbitrary descriptive labels with frames (and other objects). Script labels then allow processors to find frames by their labels.

In order for the Java INX library to generate new pages you must put distinct script labels on frames that the Java code will need to find. The script label can be any string as long as it is unique within the appropriate scope, which is normally within a single master spread. (While it should be possible to distinguish frames on different pages with the same label within a single master spread, the current INX support library cannot do so. In addition, there is no requirement in InDesign that frame on a spread be associated with a page within the spread, so it is best to simply give each frame a unique label within the the spread.)

In the common case where you have even and odd versions of the same page design it can be useful to use suffixes like "-even" and "-odd" to make it easier to select frames based on use of a common base name, e.g. "MainFlow" as the base name "MainFlow-even" and "MainFlow-odd" as the labels on the even and odd versions of the frames.

It is only necessary to put script labels on frames that will need to be selected. However, for complex page layouts and corresponding generation logic it can be helpful to put descriptive labels on all frames to aid in debugging or simply to make the intent of the page design clearer to observers.

Configuring Page Masters

InDesign page masters define the static components of pages and can also define frames that should be cloned into new pages created from the page master. You can use this feature to simplify the generation of new pages using the Java INX support library.

Each frame on a master spread can be flagged as to whether or not it should override the master frame and thus be cloned to new pages. When you create new pages interactively in InDesign, the flagged frames are automatically copied to the new page. The INX support library can do the same thing, automatically cloning master frames flagged as overridable in the page master. Following the cloning, the Java code can then use script labels on the cloned frames to find the right frame for further processing (e.g., to place a story or graphic into).

When you set up your page masters you must ensure that only those master frames that should be overridden are flagged as overridable. Any frame that should be the same on ever page with that master should be "locked" (set as not overridable) in the page master.

Using Template Frames

InDesign does not have a formal concept of "template frames". However, it is common design practice to create specific frames for use as templates, placing them outside the boundaries of the pages on a master page sequence. If you have frame templates that are not specific to a particular master page it usually makes sense to create a separate master spread just for template frames. This allows you to position the frames at the appropriate places on their pages, removing the need for the InDesign generation code to know how to position the frame when a frame's position on a page is invariant. When the position of a frame is variable then either the InDesign generation code needs to know how to position it or you have to leave the positiioning for a human to handle as a post-generation task.

A frame template normally reflects a specific size, object style, default content (if applicable) and optionally a specific location along one or more placement axes (horizontal, vertical, or stacking (z axis). For example, you might want to have a specific frame style for tip-type notes. You could design the frame, giving it specific properties and a specific object style, and give it a descriptive script label. That frame could then be cloned during InDesign generation as needed. Note that the Java code needs to know the

In manual design practice, Designers simply copy and paste these frames as needed to create new pages or page masters. The Java INX support library can do the same thing by locating frames by script label and cloning them into pages at the appropriate location.

There are two ways to set up and use frame templates:

1. Place template frames specific to a given page master outside the page boundaries of the master pages and let the InDesign generation code position the frames as appropriate. This requires custom Java code to do the positioning (as of version 0.9.11).
2. Create separate master spreads specifically for template frames and position the frames within the page boundaries as a appropriate. This allows the inDesign generation code to simply clone the frame onto the output page without modifying its original geometry, which allows for simpler Java code.

In both cases you must of course use script labels to enable the InDesign generation process to find the appropriate template.

Note also that there is no general way for the InDesign generation process to know where on a page a given paragraph will occur within flowed content that spans multiple pages. For the first page of a flow you can usually know or guess reasonably accurately where things will fall, depending on the page design details. But after that the best you can do is generate frames that have the appropriate properties for a human or InDesign script to place based on the final flow of the text.

The Java code can generate anchored frames, so that Designers can know what place in the content a given frame is associated with.

This is an Appendix

Short description for the appendix.

Appendix content goes here.

Colophon

This document uses the map, topic, and domain specializations of the DITA For Publishers project, *dita4publishers.sourceforge.net*. The various renderings are generated using the DITA Open Toolkit with extentions and customizations provided as part of the DITA For Publishers materials.

Index

A

ant parameters
    cover.graphic.file *19*
    html2.file.organization.strategy *25*
    html2.generate.dynamic.toc *25*
    html2.generate.frameset *25*
    html2.generate.static.toc *26*
    kindlegen.executable *21*

C

classification domain *33*
cover
    generating *19*
    graphic for *16*
cover.graphic.file ant parameter *19*
css
    custom *15*

D

DITA-to-InDesign *64*
domain modules *32*
domains
    classification *33*
    enumeration *33*
    formatting *33*
    publication content *33*
    topic *33*
    verse *33*

E

elements in context
    mapping to InDesign styles *65*
enumeration
    customizing for HTML output *24*
enumeration domain *33*
EPUB
    generating
        with OxygenXML *14*

F

formatting domain *33*
frameset
    generating *25*

G

graphics
    cover *16*

H

HTML
    generating *22*
HTML2 Toolkit plugin *22*
html2.file.organization.strategy ant parameter *25*
html2.generate.dynamic.toc ant parameter *25*
html2.generate.frameset ant parameter *25*
html2.generate.static.toc ant parameter *26*

I

InCopy
    generating articles from maps and topics *65*
InDesign
    generating from DITA *62*
    optimizing for automation *68*
    transformation framework *62*
InDesign styles
    mapping elements in context to *65*
installing toolkit plugins *12*

K

kindle book
    generating *21*
Kindle book
    generating
        with OxygenXML *14*
kindlegen *13*
kindlegen.executable ant parameter *21*

M

map-driven processing
    generating HTML *22*
metadata
    creating *16*
    OPF *16*
MS Word
    to-DITA transformation framework *40*

O

Open Toolkit plugins
    Word-to-DITA transform *26*
OPF metadata
    creating *16*
output file organization strategy
    generating *25*
overriding base processing *17*
OxygenXML
    generating EPUBs with *14*
    generating Kindle books with *14*
    running Word-to-DITA transform with *44*

P

parameters
    ant

cover.graphic.file *19*
html2.file.organization.strategy *25*
html2.generate.dynamic.toc *25*
html2.generate.frameset *25*
html2.generate.static.toc *26*
kindlegen.executable *21*
publication content domain *33*

R

release notes
Version 0.9.10 *10*
Version 0.9.11 *9*
Version 0.9.12 *9*
Version 0.9.13 *9*
Version 0.9.14 *9*
Version 0.9.15 *8*
Version 0.9.16 *8*
Version 0.9.18 *6*
Version 0.9.9 *10*

T

table of contents
generating *25*, *26*
toolkit plugins
installing *12*
topic domains *33*
topics
generating InCopy articles from *65*
transformation parameters
word-to-dita transformation *60*
transformation scenario *14*
troubleshooting *55*
tutorials
word-to-DITA video tutorial *57*
Typefi *62*

V

verse domain *33*
vocabulary *27*
vocabulary modules
domain *32*

W

Word, *See* MS Word
word-to-dita Toolkit plugin
Ant script for *43*
word-to-DITA transform
video tutorial *57*
Word-to-DITA transform
from OxygenXML *44*
troubleshooting *55*
Word-to-DITA transformation framework
Open Toolkit plugin for *26*