

Writing documentation with DocBook-XML 4.3



A documentation system for print and Web

John Shipman

2011-12-20 15:53

Abstract

Describes a system for writing general documentation for presentation in both Web and PDF form.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Advantages of DocBook	2
2. Relevant online files	3
3. Setting up your directory for DocBook	4
4. Creating and translating your document	4
4.1. What is DocBook and how does it relate to XML?	4
4.2. The DocBook workflow cycle	5
5. Overall section structure	6
5.1. The <code>titleabbrev</code> element: Short title	7
6. Ordinary prose paragraphs: <code>simpara</code> and <code>para</code>	8
7. Inline markup	8
8. Links: connecting your document to itself and elsewhere	11
8.1. The <code>link</code> and <code>xref</code> tags: Linking within your document	11
8.2. The <code>ulink</code> tag: Linking to a Web page	13
9. Special paragraph shapes	13
9.1. Bullet lists: <code>itemizedlist</code>	13
9.2. Numbered lists: The <code>orderedlist</code> element	13
9.3. Procedures	14
9.4. Question-and-answer sets	16
9.5. Definition lists: <code>variablelist</code>	17
9.6. Notes, warnings, cautions, etc.	18
9.7. Block quotations	19
10. Verbatim displays	19
10.1. Callouts in verbatim displays	20
10.2. Poetry	21
11. Including graphic images	21
11.1. Formal and informal figures	21
11.2. Tuning graphics for different roles	22
11.3. Scaling a figure	23

¹ <http://www.nmt.edu/tcc/help/pubs/docbook43/>

² <http://www.nmt.edu/tcc/help/pubs/docbook43/docbook43.pdf>

11.4. Inline graphics	24
11.5. How to get screen shots (Windows, MacOS, and Linux)	24
12. Tables	25
12.1. Ruled lines in tables	27
12.2. Controlling table dimensions	28
12.3. Controlling alignment in tables	28
12.4. Horizontal (column) spanning in tables	29
12.5. Vertical (row) spanning in tables	30
13. Including <i>TeX</i> and <i>LaTeX</i> math	31
13.1. Preparing a formula with <i>LaTeX</i>	31
13.2. Preparing a formula with <i>TeX</i>	32
13.3. Processing your math files for inclusion	32
13.4. Automating math display production with your <i>Makefile</i>	34
13.5. Simple inline math	35
13.6. Inline math using <i>LaTeX</i> or <i>TeX</i>	36
14. User-defined entities	36
15. Breaking your document into multiple files	38
16. Decluttering the project directory	39
17. Literate programming with DocBook	39
17.1. Controlling spurious blanks and blank lines	40
18. Special characters	41
18.1. Universally available entities	41
18.2. International character entities	41
18.3. The Greek alphabet	43
18.4. Special symbols	44
19. Model files for <i>make</i>	46
19.1. <i>make-basic</i> : A basic <i>Makefile</i>	47
19.2. <i>make-lit</i> : A <i>Makefile</i> for literate programming	48
19.3. <i>make-large</i> : A <i>Makefile</i> for large projects	51
20. FOP: An older, free toolchain	53
20.1. FOP limitations	53
20.2. Bad page breaks	54
20.3. Using tables inside <code><listitem></code>	54
20.4. Graphics file support	54
21. Converting DocBook-SGML 4.1 documents	54
22. Converting DocBook 3.0 documents	54

1. Advantages of DocBook

The DocBook system has these advantages over other methods of creating documentation:

- The same document can be translated mechanically to both Web-based and printable formats.
- You as an author can concentrate on the content of your document, without worrying about how it will appear.

The Tech Computer Center supplies a locally-customized installation of the DocBook translation software for output to PDF or Web form.

- The Modular Style Sheets web site³ provides a base set of HTML and PDF stylesheets. You may use these as the basis for your own customizations.

³ <http://docbook.sourceforge.net/projects/xsl/>

- *Customization of the 4.3 DocBook XSL Stylesheets for the TCC*⁴ describes how the Modular Style Sheets were customized for this location.

The mechanical translation to various formats can be improved and tuned independently of the writing process. These improvements do not at all affect the DocBook files you write.

Note

This document assumes you are using DocBook-XML revision 4.3. If your document was done under the SGML versions, see Section 22, “Converting DocBook 3.0 documents” (p. 54) and Section 21, “Converting DocBook-SGML 4.1 documents” (p. 54) below.

2. Relevant online files

A number of files referenced in this document are available online.

- `model.xml`⁵: A starter DocBook XML file.
- `make-basic`⁶: A starter Makefile file. See Section 19.1, “make-basic: A basic Makefile” (p. 47).
- `make-lit`⁷: Variant Makefile for lightweight literate programming; see Section 19.2, “make-lit: A Makefile for literate programming” (p. 48).
- `make-large`⁸: Variant Makefile to implement the technique described in Section 16, “Decluttering the project directory” (p. 39); see Section 19.3, “make-large: A Makefile for large projects” (p. 51).
- `logo.png`⁹ is the TCC logo, to be used on Web pages. If you are not writing an official TCC document, you can replace this with your own artwork, 2” wide, with a transparent background.
- `logo.jpg`¹⁰ is the JPG version of the TCC logo, to be used in PDF output. May be replaced by your own artwork, 2” wide.
- `docbook43.xml`¹¹: The source file for the document you are now reading, this file contains examples of every construct including tables, figures, and mathematical equations.

Note

If you bring up this file in a browser, what you see will not be the exact source. In particular, the internal entities (see Section 14, “User-defined entities” (p. 36)) will be replaced by their equivalent text. Use *File* → *Save Page As* to get an exact copy.

- The Makefile¹² for this document.

⁴ <http://www.nmt.edu/tcc/doc/docbook43/ims/web/>

⁵ <http://www.nmt.edu/tcc/help/pubs/docbook43/model.xml>

⁶ <http://www.nmt.edu/tcc/help/pubs/docbook43/make-basic>

⁷ <http://www.nmt.edu/tcc/help/pubs/docbook43/make-lit>

⁸ <http://www.nmt.edu/tcc/help/pubs/docbook43/make-large>

⁹ <http://www.nmt.edu/tcc/help/pubs/docbook43/logo.png>

¹⁰ <http://www.nmt.edu/tcc/help/pubs/docbook43/logo.jpg>

¹¹ <http://www.nmt.edu/tcc/help/pubs/docbook43/docbook43.xml>

¹² <http://www.nmt.edu/tcc/help/pubs/docbook43/Makefile>

3. Setting up your directory for DocBook

The DocBook software runs only under the Linux operating system. Also, the process of document creation is much, *much* easier with a validating XML editor. For an *emacs*-based validating XML editor, see *XML document authoring with emacs nxml-mode*¹³. If all else fails, you can always use a regular text editor, but you will have to type every character of every tag yourself.

The *make* system is a great timesaver in carrying out the steps of the DocBook document development cycle. Refer to the *man* page for *make* if you are not familiar with this product.

Here is the procedure for setting up your directory:

1. Create the directory with **mkdir** if it does not already exist.
2. Use the **cd** command to move to the directory.
3. See Section 2, “Relevant online files” (p. 3) and download `make-basic`, `model.xml`, `logo.png`, and `logo.jpg` to the current directory.
4. Invent a name for your DocBook file that ends in `.xml`, and rename `model.xml` as that file.

For example, if your document is about dust abatement, you might call it `dust.xml`. The rename command for this example would be:

```
mv model.xml dust.xml
```

5. Rename the `make-basic` file as `Makefile`:

```
mv make-basic Makefile
```

Then, in `Makefile`, find this line:

```
BASENAME = your-document-base-name-here
```

and replace the part after the “=” with the name you gave your document in the previous step, without the `.xml` part.

For the example in the previous step, this line would now read:

```
BASENAME = dust
```

You will develop your document by editing the `.xml` file as described in later sections.

4. Creating and translating your document

4.1. What is DocBook and how does it relate to XML?

DocBook is a document type derived from an ISO standard called XML (eXtensible Markup Language)¹⁴. (HTML, used to write World Wide Web pages, is also a markup language, and XML is similar in syntax.) All XML document types use the idea of *tags* to structure a document.

- A tag always starts with a less-than character (“<”) and ends with a greater-than character (“>”).
- Most tags are used in pairs, with content enclosed between a *start tag* and an *end tag*.

¹³ <http://www.nmt.edu/tcc/help/pubs/nxml/>

¹⁴ <http://www.xml.com/axml/testaxml.htm>

For example, major sections of a document are always enclosed between a `<section>` start tag and a `</section>` end tag.

- The term *element* refers to a pair of tags and everything in between them.
- A start tag has the form

```
<tag-name attributes>
```

and the end tag has the form

```
</tag-name>
```

where the *tag-name* is some name that describes what the tag does.

- In some cases, you can specify optional *attributes* that modify the characteristics of the element. Each attribute has this form:

```
N="V"
```

or

```
N='V'
```

where *N* is the attribute's name and *V* is the attribute's value.

Here is an example of a start tag with attributes:

```
<section id="intro" status='draft'>
```

- An *empty element* is a special element consisting of one tag. In this case, the tag must have a forward slash (/) just before the closing `>`. For example, the `<colspec/>` empty element describes the format of a table column in a table (see Section 12, "Tables" (p. 25)).
- You can add whitespace (spaces, tabs, or line breaks) anywhere before or after an attribute, or before the closing `>`.

4.2. The DocBook workflow cycle

Here is the basic workflow for the creation and maintenance of a document using DocBook.

1. Use a validating XML editor or other editor to add the content of your document to the XML file.
2. Follow the directions given in comments in the skeleton file. You will need to fill in the document's overall title, information on who you (the author) are, and so on.
3. Once the model is filled in, add the title of the first section where indicated, and begin writing the body of the document. The writing process mainly proceeds by:
 - Adding XML *tags* that describe the structure of the document.
 - Placing the content of the document inside these tags.
4. To render it into Web form as a group of HTML files, use:

```
make web
```

To translate your document into one of the final forms, use the *make* utility, which is driven by the *Makefile* you prepared above in Section 3, "Setting up your directory for DocBook" (p. 4).

To render into print form using PDF (Adobe Page Description Format) file, use the Unix command:

```
make pdf
```

Or just type a **make** command by itself with no arguments to build both:

```
make
```

5. Overall section structure

Here is the model file discussed in Section 3, "Setting up your directory for DocBook" (p. 4):

model.xml

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd">
<article>
  <!-- Replace all fields ### below. Then delete this comment-->
  <articleinfo >
    <title >###
  </title >
    <authorgroup >
      <author >
        <firstname >###
      </firstname >
        <surname >###
      </surname >
      </author >
    </authorgroup >
    <address ><email >###
  </email >
    </address >
    <revhistory>
      <revision>
        <!--Place revision information here: RCS tags, etc.-->
        <revnumber>###</revnumber>
        <date>###</date>
      </revision>
    </revhistory>
    <abstract>
      <para>
        <!--All TCC documents must have an abstract.-->
        ###
      </para>
      <para>
        This publication is available in <ulink url="###"
        >Web form</ulink > and also as a <ulink
        url="###.pdf"
        >PDF document</ulink >. Please
        forward any comments to <userinput
        >tcc-doc@nmt.edu</userinput >.
      </para>
    </abstract>
  </articleinfo >
  <section id='intro' >
```

```

    <!-- On the next line, place the title of your first section.-->
    <title >###
</title >
    <!-- Add the body of your first section here.-->
    </section >
    <!--Place additional sections here.-->
</article >

```

Note the last few lines above. This is the skeleton of the first section of your document. Place the section's title inside the `title` element, and add the section's content after it.

Each `section` element is a top-level section. They will be numbered as sections 1, 2, 3, and so on. To add subsections, use a `section` element inside the `section` element.

For example:

```

<section id='intro'>
  <title>Main section title</title>
  <para>Main section content...</para>
  <section id='sir-gawain'>
    <title>First subsection title</title>
    <para>First subsection content...</para>
  </section>
  <section id='sir-robin'>
    <title>Second subsection title</title>
    <para>Second subsection content...</para>
  </section>
</section>

```

If the main section were section 3, then the two subsections inside it would be numbered 3.1 and 3.2.

Important

You must invent a unique identifier for each section, and attach an `id='I'` attribute to the `section` element. You will use these identifiers to generate automatic internal cross-references in your document; see Section 8.1, “The `link` and `xref` tags: Linking within your document” (p. 11). Identifiers may contain any combination of letters, hyphens (-), underbars (_), or digits.

To help you manage larger documents, there is a utility that displays all the section identifiers for one DocBook document in an outline format. See *docbookindex: ID indexer for DocBook*¹⁵. As an example, here is the `toc.pdf` file¹⁶ showing all the section identifiers for this document.

You can include more `section` elements for sub-sub-sections, and so on.

5.1. The `titleabbrev` element: Short title

The `title` element inside your `articleinfo` group will appear in the running footer on every page in the PDF version of the document. If your title is too long, it will be folded into two or more lines inside the footer.

To cure this problem, place a `titleabbrev` element just after the main `title` element, containing a shorter version of the title that will fit inside the running footer. For example:

¹⁵ <http://www.nmt.edu/tcc/projects/docbookindex/>

¹⁶ <http://www.nmt.edu/tcc/help/pubs/docbook43/toc.pdf>

```

<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd">
<article >
  <articleinfo >
    <title >
      Exegesis of archetypal content and pataphysical normatives in
      <citetitle >Monty Python and the Holy Grail</citetitle >
    </title >
    <titleabbrev >
      <citetitle >Monty Python</citetitle > and
      pataphysics
    </titleabbrev>
    ...
  </article>

```

6. Ordinary prose paragraphs: `simpara` and `para`

For everyday text paragraphs, enclose each paragraph in a `simpara` element. Here's an example from the *Declaration of Independence*:

```

<simpara>
  He has abdicated Government here, by declaring
  us out of his Protection and waging War against us.
</simpara>
<simpara>
  He has plundered our Seas, ravaged our Coasts, burnt
  our Towns, and destroyed the Lives of our People.
</simpara>

```

For paragraphs which include other complex structures such as bullet lists, programlisting shots, and such, it is necessary to use the `para` element. These will look the same in the final form as a `simpara` paragraph.

7. Inline markup

DocBook has a number of elements that are useful for marking up content within a paragraph. By *inline markup*, we refer to tags that appear within a paragraph and do not change or interrupt its basic paragraph shape.

acronym

An abbreviation, generally made from the initial letters of words, and sometimes pronounceable. For example:

```

<acronym>PEBKAC</acronym> stands for: Problem exists
between keyboard and chair.

```

PEBKAC stands for: Problem exists between keyboard and chair.

application

Names of packages. Example:

```

Start <application>emacs</application>.

```

Start *emacs*.

citetitle

Used for citing works by their title. Example:

```
Have you read <citetitle>Ringworld</citetitle>?
```

Have you read *Ringworld*?

code

For program source code. Example:

```
Next we define the <code >panic()</code > function.
```

Next we define the `panic()` function.

computeroutput

Computer-generated output. Example:

```
<computeroutput>0A210I 0LD PSW WAS FF04230C 1200234B</computeroutput>
```

0A210I 0LD PSW WAS FF04230C 1200234B

emphasis

Used for emphasized text. Example:

```
Don't <emphasis>do</emphasis> that then.
```

Don't *do* that then.

For boldface, add the attribute `role="strong"`. Example:

```
A five-ounce bird could <emphasis role="strong">not</emphasis>  
hold a one-pound coconut.
```

A five-ounce bird could **not** hold a one-pound coconut.

filename

For file names and path names. Example:

```
Packages live in <filename>/fs/packages</filename>.
```

Packages live in `/fs/packages`.

firstterm

For the first use of a term just being defined. Example:

```
The term <firstterm >spam</firstterm > refers to unsolicited  
commercial e-mail.
```

The term *spam* refers to unsolicited commercial e-mail.

foreignphrase

Used to set off words in a different language than the surrounding text.

```
These sharks are in genus <foreignphrase>Squalus</foreignphrase>.
```

These sharks are in genus *Squalus*.

guibutton

Buttons in a graphical user interface. Example:

```
To exit, click on the <guibutton>Quit</guibutton> button.
```

To exit, click on the *Quit* button.

guicon

An icon in a graphical user interface.

guilabel

A label in a graphical user interface.

guimenu

A menu in a graphical user interface. Example:

```
Pull down the <guimenu>Team</guimenu> menu.
```

Pull down the *Team* menu.

keysym

For names of keys on a keyboard. Example:

```
Press <keysym>Enter</keysym>.
```

Press *Enter*.

quote

For in-line quotations. One could simply use the double-quote characters "...", but the advantage of the `quote` element is that it correctly handles quotes within quotes. For example:

```
I said, <quote>As Carol Schaffer used to say,
<quote>common sense isn't</quote>.</quote>
```

I said, "As Carol Schaffer used to say, 'common sense isn't'."

replaceable

Use this tag for parts of a template, pattern, or general case that will be replaced with specific items in practice. Example:

```
Call your input file <filename><replaceable
>f</replaceable >.tex</filename >, where
<filename ><replaceable >f</replaceable
></filename > is some name of your choice.
```

Call your input file *f.tex*, where *f* is some name of your choice.

sgmltag

For the display of any XML or SGML tag. Use the attribute `class="starttag"` for a starting tag, and `class="endtag"` for an ending tag. Example:

```
Enclose the page title within <sgmltag class="starttag"
>title</sgmltag >...<sgmltag class="endtag"
>title</sgmltag > tags.
```

Enclose the page title within `<title>...</title>` tags.

subscript

Displays the contents below the baseline. Example:

```
Drink more H<subscript>2</subscript>O.
```

Drink more H₂O.

superscript

Displays the contents above the baseline. Example:

```
Celebrate the 4<superscript>th</superscript> of July.
```

Celebrate the 4th of July.

userinput

For commands or other user input. Example:

```
Type the command <userinput>make trouble</userinput>.
```

Type the command **make trouble**.

varname

For variable names in programs. Example:

```
Add one to <varname>sheepCount</varname>.
```

Add one to sheepCount.

8. Links: connecting your document to itself and elsewhere

The advent of hypertext technology gives us the opportunity to make it much easier for the reader to jump around when looking for information. However, DocBook's ability to create both Web pages and paper copy complicates the task of linking. We want the Web versions to be hot-clickable, but keep in mind that the paper version must still be useful even though we can't click on it.

The paper equivalent of a hyperlink is a cross-reference. A cross-reference within the same work might say something like "See the section on frog identification below." If you phrase your document in this way and use the right tags carefully, this will read normally in the print version and the Web version will read the same way and also be a link.

- For cross-references to other locations in the same document, refer to Section 8.1, "The `link` and `xref` tags: Linking within your document" (p. 11).
- For references to a Web page outside the document, see Section 8.2, "The `ulink` tag: Linking to a Web page" (p. 13).
- Both Web and paper documents may refer to external works: printed books or documents. For this kind of linking, use the `citetitle` element; see Section 7, "Inline markup" (p. 8).

8.1. The `link` and `xref` tags: Linking within your document

To refer to another location in the same document:

1. Invent a unique identifier name for the location to which you want to refer. The first character must be a letter, and the rest letters, numbers, hyphens, and the underscore (`_`) character.

For example, you might use the identifier `flute-tuning` for a section on how to tune a flute.

2. Find the start tag for the element to which you want to refer, and attach an `id="I"` attribute to that tag, where `I` is the identifier name you invented in the previous step. Any DocBook start tag can carry an `id` attribute.

For example, to refer to a main section, add the `id` attribute to the `section` element.

3. At the location where you want to cross-refer, use a tag of the form:

```
<link linkend="I">T</link>
```

where *I* is the identifier defined in the previous step and *T* is the text of the link.

For example:

```
<section id="thule-history">
  <title>The History of Thule</title>
  ...
<section>
  ...
  <para>For more amusing folk traditions of Greenland, see
    <link linkend="thule-history">History of Thule</link> above.
  </para>
```

- In the HTML output, this will produce a clickable link.
- In the PDF output, the content of the `link` element will also be clickable, although it will not appear any differently than the surrounding text.

You can also use the empty element `<anchor id="I" />` to define a target location, where *I* is the same unique identifier.

You may find it easier to use the `xref` element to link to other locations within your document. This is an empty element; the link text is generated automatically.

There are three ways to use this element. All three require that you assign a unique `id` attribute to the element to which you are linking:

- Just encode it as `<xref linkend="I" />`, where *I* is the target identifier. The link text will depend on the type of element. For example, a reference to a `chapter` element will generate link text containing the chapter number and title.
- If there is an element somewhere near the target identifier whose content you would like to use as the link text, use this construct:

```
<xref linkend="I" endterm="J" />
```

where *I* is the identifier of the element you want to link to, and the link text will be taken from the content of the element with `id="J"`.

For example, suppose a chapter starts like this:

```
<chapter id="ch11">
  <title>The <emphasis id="ch11-abbr">Last
    Chapter</emphasis></title>
```

Then this reference would use “Last Chapter” as the link text:

```
<xref linkend="ch11" endterm="ch11-abbr" />
```

- You can specify what link text you want to use for a specific target by adding an `xreflabel` attribute to the target element. Any `xref` element that refers to that target will automatically use that attribute's value as its link text.

For example, suppose you start a chapter like this:

```
<chapter id="ch14" xreflabel="The Silly Chapter">...
```

then any reference of the form `<xref id="ch14" />` would use “The Silly Chapter” as its link text.

8.2. The `uLink` tag: Linking to a Web page

To refer to a location on a Web page outside the current document, use the `uLink` element with a `url` attribute that specifies the URL (Uniform Resource Locator) of the target.

- In the HTML output, this will produce a clickable link.
- In the PDF output, DocBook will place the URL in a footnote.

Example:

```
See the <uLink url="http://www.nmt.edu/tcc/help/">TCC Help
System</uLink>.
```

would display as: "See the TCC Help system¹⁷."

9. Special paragraph shapes

9.1. Bullet lists: `itemizedlist`

A *bullet list* is a set of narrower paragraphs, each shown with a round dot (the *bullet*) before it. Typical uses are lists of features, lists of important points, and such.

Enclose the entire bullet list inside an `itemizedlist` element. Then, enclose each item within a `listitem` element containing one or more `para` or `simplpara` elements.

For example, this input:

```
<itemizedlist>
  <listitem>
    <para>Mangos.</para>
  </listitem>
  <listitem>
    <para>Chirimoya.</para>
  </listitem>
</itemizedlist>
```

produces this output:

- Mangos.
- Chirimoya.

Normally, bullet lists have a fairly generous amount of space between the bullets. To reduce this space, add a `spacing="compact"` attribute to the `itemizedlist` element. Here is the example list again in compact spacing:

- Mangos.
- Chirimoya.

9.2. Numbered lists: The `orderedlist` element

To produce a numbered list of items, use the `orderedlist` element. Here's an example:

1. DocBook renders documents in both print and Web form.

¹⁷ <http://www.nmt.edu/tcc/help/>

2. It frees the author from most concerns of presentation.

Here's how the above list looks in source form:

```
<orderedlist>
  <listitem>
    <para>
      DocBook renders documents in both print and Web form.
    </para>
  </listitem>
  <listitem>
    <para>
      It frees the author from most concerns of presentation.
    </para>
  </listitem>
</orderedlist>
```

There are a number of attributes you can add to the `orderedlist` element to change the way the entries are numbered.

numeration

The default value is `numeration="arabic"`, with entries numbered 1, 2, 3, Other possible values include:

- `numeration="loweralpha"` for a, b, ...
- `numeration="lowerroman"` for i, ii, iii, ...
- `numeration="upperalpha"` for A, B, C, ...
- `numeration="upperroman"` for I, II, III, ...

inheritnum="inherit"

Normally, when you use an ordered list inside another ordered list, the inner list has its own numbering. However, if you specify `inheritnum="inherit"`, each item number in the inner list will have the item number from the next outer list prepended to it.

For example, if an inner list is under an outer-list item numbered 17, and the inner list had the `inheritnum="inherit"` attribute, its items would be numbered 17.1, 17.2, 17.3,

spacing="compact"

Normally the items of an ordered list are separated by generous amounts of space. Use the `spacing="compact"` attribute to squeeze out most of the space between the items.

continuation="continues"

Sometimes you have to break an ordered list into multiple pieces, with text paragraphs or other material between the pieces. This attribute allows you to start the numbering of a new list where the old one left off.

For example, suppose the first chunk of an ordered list ended with item XIV, and another ordered list followed it, starting with:

```
<orderedlist numeration="upperroman" continuation="continues">
```

That second list would start with item XV.

9.3. Procedures

A step-by-step procedure is generally presented in a form similar to the bullet list, but with step numbers taking the place of bullets.

Enclose the entire procedure within a `procedure` element. Each step is enclosed within a `step` element containing one or more `para` or `sipara` elements.

For example, this input:

```
<procedure>
  <step>
    <para>
      Throw the football.
    </para>
  </step>
  <step>
    <para>
      Pick it up first.
    </para>
  </step>
</procedure>
```

produces this output:

1. Throw the football.
2. Pick it up first.

If the steps of your procedure refer to other steps, you can get DocBook to insert the step number into each reference automatically. To do this:

1. Invent a name *I* for each step, and add an attribute `id=I` to the corresponding `step` element.
2. Wherever you want to refer to a step, use the element `<xref linkend="I" />`, where *I* is the name of the step you are referring to.

This element will be replaced by the text "step N" where N is the step number.

Here's an example. This is a (facetious) solution to the famous Halting Problem. First, in output form:

1. Has the program halted? If so, go to Step 3 (p. 15).
2. Go to Step 1 (p. 15).
3. Done: we now know that the program halts.

And now the source for the above:

```
<procedure>
  <step id="start-step">
    <para>
      Has the program halted? If so, go to
      <xref linkend="done-step" />.
    </para>
  </step>
  <step id="loop-step">
    <para>
      Go to <xref linkend="start-step" />.
    </para>
  </step>
  <step id="done-step">
    <para>
      Done: we now know that the program halts.
    </para>
  </step>
</procedure>
```

```
</para>
</step>
</procedure>
```

9.4. Question-and-answer sets

DocBook supports sets of questions and answers, such as Frequently Asked QUestions (FAQ) documents.

The top-level element for a set of questions and answers is `qandaset`. Here is a very simple set of two Q&A pairs:

At the Bridge of Death

Q: What is your favorite color?

A: Blue.

Q: What is the air-speed velocity of an unladen swallow?

A: What do you mean? An African or European swallow?

Here's how that list is encoded:

```
<qandaset defaultlabel='qanda'>
  <title>At the Bridge of Death</title>
  <qandaentry>
    <question>
      <para>
        What is your favorite color?
      </para>
    </question>
    <answer>
      <para>
        Blue.
      </para>
    </answer>
  </qandaentry>
  <qandaentry>
    <question>
      <para>
        What is the air-speed velocity of an unladen swallow?
      </para>
    </question>
    <answer>
      <para>
        What do you mean? An African or European swallow?
      </para>
    </answer>
  </qandaentry>
</qandaset>
```

- To specify the format, in the `qandaset` element, include a `defaultlabel` attribute with one of these values:

<code>defaultlabel='qanda'</code>	Label the questions with "Q:" and the answers with "A:".
-----------------------------------	--

<code>defaultlabel='number'</code>	Number the entries.
<code>defaultlabel='label'</code>	Each <code>question</code> element and each <code>answer</code> element must have a <code>label</code> attribute containing the string to be used as a label.

- For a simple list of questions and answers, enclose each item in a `qandaentry` element.
- Inside the `qandaentry`, enclose each question in a `question` element and each answer in an `answer` element.

You can have any number of `question` elements in a `qandaentry` element.

You can have any number of `answer` elements in a `qandaentry` element, even none at all (for questions with no answer).

If you want to structure your Q&A set into sections, you can use a `qandadiv` element inside your `qandaset` element, with one or more `qandaentry` elements inside that.

To give the section a title, use a `title` element after the `qandadiv` element.

Here's a partial example:

```
<qandaset defaultlabel='qanda'>
  <title>Frequently axed questors</title>
  <qandadiv>
    <title>Trolls I have put down</title>
    <qandaentry>
      ...
    </qandaentry>
    ...
  </qandadiv>
  <qandadiv>
    <title>Castle Anthrax</title>
    <qandaentry>
      ...
    </qandaentry>
    ...
  </qandadiv>
</qandaset>
```

You can even use `qandadiv` elements inside other `qandadiv` elements to divide your list into sublists, sub-sublists, and so on.

9.5. Definition lists: `variablelist`

Another commonly used special paragraph shape is to display a list of terms to be defined, each followed by an indented paragraph containing the description.

For this shape:

- Enclose the entire list in a `variablelist` element.
- Each term-definition pair is enclosed in a `varlistentry` element.
- Inside each `varlistentry` element, enclose the term to be defined inside a `term` element, and enclose the paragraph or paragraphs defining the term inside a `listitem` element.

Here is an example of a list containing two terms:

```

<variablelist>
  <varlistentry>
    <term>
      <code >.__abs__(self)</code >
    </term>
    <listitem>
      <para>
        Defines the behavior of the <code >abs()</code > function
        when applied to an object of your class.
      </para>
    </listitem>
  </varlistentry>
  <varlistentry>
    <term>
      <code >.__add__(self,<replaceable >other</replaceable
      >)</code >
    </term>
    <listitem>
      <para>
        Defines the behavior of this class for <code
        >self+<replaceable>other</replaceable ></code >.
      </para>
    </listitem>
  </varlistentry>
</variablelist>

```

And here is how that looks when formatted:

- **.__abs__(self)**
 Defines the behavior of the `abs()` function when applied to an object of your class.
- **.__add__(self, other)**
 Defines the behavior of this class for `self+other`.

9.6. Notes, warnings, cautions, etc.

Sometimes you need to make a block of text stand out against the background. DocBook has several elements for such purposes:

- `note`
- `warning`
- `important`
- `caution`
- `tip`

Typically you will put one or more regular paragraphs (using the `para` element) inside such elements. Here's an example:

Warning

Do not touch the electric fence!

And here's that warning in source form:

```
<warning>
  <para>
    Do not touch the electric fence!
  </para>
</warning>
```

9.7. Block quotations

Two different elements are used to present quotations set off in separate blocks:

- Use `epigraph` when the quotation starts a new chapter or section.
- Use `blockquote` for quotations elsewhere.

In either case, the content of the element consists of an optional `attribution` element that tells the reader who said this. The actual quotation follows, typically wrapped in one or more `para` elements. Generally the attribution will be presented *after* the quotation.

Here's an example:

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

—C. A. R. Hoare

The source for the quote looks like this:

```
<blockquote>
  <attribution>C. A. R. Hoare</attribution>
  <para>
    There are two ways of constructing a software
    design. One way is to make it so simple that there are
    obviously no deficiencies. And the other way is to make
    it so complicated that there are no obvious
    deficiencies.
  </para>
</blockquote>
```

10. Verbatim displays

By *displays*, we mean the presentation of information in some form other than a paragraph: illustrations, representations of computer screens, and so on.

To present one or more lines in a monospaced (fixed-width) font, as in a computer program, enclose the lines within a `programlisting` element.

For example, this input:

```
<programlisting>
  10 PRINT "BASIC IS OVER 40 YEARS OLD"
  20 GOTO 10
  30 END
</programlisting>
```

produces this output:

```
10 PRINT "BASIC IS OVER 40 YEARS OLD"
20 GOTO 10
30 END
```

You can also use the `screen` element to display something that appears on a screen. The formatting is the same as for the `programlisting` element.

Line breaks are preserved starting immediately after the start tag. Therefore, if you start your code display on the line following that tag, the display will start with a blank line. There is a way to avoid this: move the closing ">" of the start tag to the beginning of the first line of code. Here is the above example formatted so as to eliminate the initial blank line:

```
<programlisting
> 10 PRINT "BASIC IS OVER 40 YEARS OLD"
  20 GOTO 10
  30 END
</programlisting>
```

10.1. Callouts in verbatim displays

Sometimes you want to present a program listing or screen shot with *callouts*, little numbered graphic tags that appear within the display. Then, following the display, you present textual discussions for each callout. Here's an example:

```
AWAKE! for Morning in the Bowl of Night 1
Has flung the Stone that puts the Stars to Flight:
And Lo! the Hunter of the East has caught
The Sultan's Turret in a Noose of Light. 2
```

- 1** Note the gratuitous capitalization.
- 2** It appears that someone has struck the Sultan on the turret with an alarm clock.

In order to use callouts, you must have a subdirectory named `callouts` in the same directory as your document, containing the actual callout images in two formats (PNG for web pages, PDF for print presentation). There are two ways to make these callouts available under Linux.

1. Make a soft link from your directory:

```
ln -s /u/www/docs/tcc/help/image/callouts .
```

2. Or, copy an archive file containing that whole directory and unpack it in your directory:

```
cp /u/www/docs/tcc/help/image/callouts.tgz .
tar -xvzf callouts.tgz
```

That directory currently contains graphics for twenty callouts numbered 1 to 20. If you don't like their appearance or need more than 20, see the `README` in that directory to see how to create your own.

Once you have installed the callout graphic files, to use callouts in your `programlisting` or `screen` element:

1. Add an element of the form `<co id="I"/>` within the display where you want a callout to appear. Invent a unique identifier *I* to be used later.

2. When you have decorated your display with `CO` elements, add a `calloutlist` element after the end of the display.
3. Within that `calloutlist` element, place one `callout` element for each `CO` element in the display. To each `callout` element, add an attribute of the form `arearefs="I"`, where the value of *I* is the value of the `id` attribute of the corresponding `CO` element.

Add your textual description (or graphics or whatever) within the `callout` element. If you need to put anything other than ordinary text inside the `callout` element, wrap it inside a `para` element.

Here's how the above example looks in source form:

```
<screen
>   AWAKE! for Morning in the Bowl of Night <co id="khay1"/>
   Has flung the Stone that puts the Stars to Flight:
   And Lo! the Hunter of the East has caught
   The Sultan's Turret in a Noose of Light. <co id="khay2"/>
</screen>
<calloutlist>
  <callout arearefs="khay1">
    Note the gratuitous capitalization.
  </callout>
  <callout arearefs="khay2">
    It appears that someone has struck the Sultan on the
    turret with an alarm clock.
  </callout>
</calloutlist>
```

10.2. Poetry

When formatting poetry and similar text, use the `literallayout` tag. Like the `programlisting` element, it presents the content with all whitespace and line breaks intact, but it uses a regular text font. Here's an example from Piet Hein:

```
Problems worthy
  Of attack
Prove their worth
  By hitting back.
```

11. Including graphic images

You can easily include images in a DocBook document. However, you may have to produce two different versions of some images so that they will look decent in both print and Web presentation.

11.1. Formal and informal figures

A *formal figure* is a graphic image presented with a title. For example, suppose you have scanned a photo and called the result `lanai.jpg`. Here is how you would include that photo as a formal figure:

```
<figure>
  <title>Lanai from the air</title>
  <mediaobject>
```

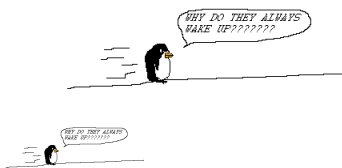
```
<imageobject>
  <imagedata fileref="lanai.jpg"/>
</imageobject>
</mediaobject>
</figure>
```

An *informal figure* is just a figure without the title. To present the same graphic as an informal figure:

```
<informalfigure>
  <mediaobject>
    <imageobject>
      <imagedata fileref="lanai.jpg"/>
    </imageobject>
  </mediaobject>
</informalfigure>
```

If you want to *change the displayed size* of the image, add an attributes `scale="P"` to the `imagedata` element, where *P* is the percentage size, e.g., 33 for one-third size.

Here is an example of a 500x100-pixel panel from Pokey the Penguin¹⁸ shown first at half size (`scale="50"`), then at one-quarter size (`scale="25"`).



11.2. Tuning graphics for different roles

For some types of graphics, such as photos or screenshots, the technique above will work fine.

However, due to the different resolutions of screens and printers, some types of graphics—especially diagrams and other vector-type drawings—will not look right both in Web and print presentation.

Consider the case of a graphic prepared using a drawing program such as *xfig*. If the graphic is exported as a JPEG or GIF image, it must be rasterized. If it rasterized at its design size, it might look okay on a screen, but in print it is likely to suffer from *aliasing*, the “stairstep” effects caused by not enough dots.

A solution that works for print presentation is to export the figure at some large magnification, like 200% or 400%, and then use `scale="50"` or `scale="25"` to shrink it down so it fits into the right space on the printed page.

However, the problem with that approach is that the graphic will now be two or four times too large in its Web form.

The solution is to prepare two different forms of the original graphic, one optimized for print presentation and one for Web use.

- For Web presentation, prepare the graphic as one of the common rasterized formats: JPEG, PNG, or GIF.
- If possible, use PDF format for print presentation. PDF format is a scalable vector representation that allows the graphic to be scaled in a way optimized for the specific printer being used to render it.

¹⁸ <http://yellow5.com/pokey/>

Once you have prepared the two forms, you can then include them in the same document as two different `imageobject` elements within the same `mediaobject` element.

In the HTML version, add to the `mediaobject` element an attribute `role="html"`. In the PDF version, add an attribute `role="fo"`.

For example, suppose you have produced a diagram in the *xfig* drawing program and called it `flow.fig`. For HTML, export it as in PNG format as `flow.png`; then export it in PDF format as `flow.pdf`. To include both figures, you might use this construct:

```
<figure>
  <title>Mill flow diagram</title>
  <mediaobject>
    <imageobject role="html">
      <imagedata fileref="flow.png"/>
    </imageobject>
    <imageobject role="fo">
      <imagedata fileref="flow.pdf"/>
    </imageobject>
  </mediaobject>
</figure>
```

Once you have this dual presentation in place, you can adjust the size of each figure independently by using `scale="..."` attributes in the `imagedata` elements.

11.3. Scaling a figure

When sizing a figure for presentation, there are two different areas involved:

- The *image size* is the size of the image itself.
- The *viewport* is the area in which the figure is supposed to be presented.

If one dimension of the image is smaller than that dimension of the viewport, the result will be empty space around the figure. But what if the viewport is smaller than the image? In print presentation, the image will be cropped; but in Web presentation the image will not be cropped.

In general there are six different attributes of `imagedata` that affect scaling:

contentwidth and contentdepth

These attributes set the desired image size. The attribute value must be a number, optionally followed by one of the dimension codes `px` (pixels, the default), `pt` (points), `cm` (centimeters), `mm` (millimeters), `pc` (picas), `in` (inches), `em` (ems), or `%` (percent, relative to the page width for horizontal dimensions).

For example, if you want the image to be 125mm×75mm, use:

```
<imagedata contentwidth="125mm" contentdepth="75mm" .../>
```

You need only specify one of these two attributes; if only one dimension is specified, the other dimension will be scaled to preserve the *aspect ratio* (ratio of height to width).

width and depth

These attributes size the viewport. If you specify the width as a percentage (%), it will be treated as a percentage of the available page width.

scale

Resizes the graphic as a percentage of the original; omit the % symbol from the attribute value. For example, to render a graphic half-size, use `scale="50"`.

scalefit

If you specify `scalefit="1"`, the graphic will be scaled to fit the available area—either the graphic size or the viewport size, whichever is filled first. If you specified no other sizing attributes, the graphic will be scaled to fit the page width.


11.4. Inline graphics

To include an inline figure in the middle of a paragraph:

1. Use the `inlinemediaobject` element.
2. Inside this element, place an `imageobject` to specify the image. The format of this element is described in Section 11.1, “Formal and informal figures” (p. 21).
3. Include a `textobject` element with text that will appear in situations where the image cannot be displayed, such as browsers for the visually impaired.

Here is an example:



Use this icon  to create a circle by specifying its diameter.

And here is the source for this paragraph:

```
<para>
  Use this icon
  <inlinemediaobject>
    <imageobject>
      <imagedata fileref="circle-2.jpg"/>
    </imageobject>
    <textobject>
      <phrase>Icon for circle by diameter</phrase>
    </textobject>
  </inlinemediaobject>
  to create a circle by specifying its diameter.
</para>
```

11.5. How to get screen shots (Windows, MacOS, and Linux)

To capture an image from your screen, follow the procedure below for your operating system.

11.5.1. How to get a Windows screen shot

On Windows 2000 and later systems, press the *PrintScrn* button. This will take a shot of the entire screen. You can then use the paste function in your favorite photo editor to make a copy of this screen shot, crop out the part you want, and so forth.

11.5.2. How to get a MacOS screen shot

To capture an image of a *specific rectangular area*, press *Apple-shift-4* (that is, while holding down the *Apple* and *shift* keys, press 4). The cursor will turn into a crosshair symbol. Move the mouse to one corner of the desired area, press and hold the left button, drag the mouse to the opposite corner of the area, and release the button. An icon named “Picture 1” (or other number) will appear in your desktop; this will contain the image of the selected area in PNG (Portable Network Graphics) format.

To capture an image of *one window*, press *Apple-shift-4*, and when the cursor turns into a crosshair symbol, press the *space* bar. The cursor turns into a picture of a camera. Move the mouse into the window you want to photograph, and that window will turn slightly gray. Click the left button and the image icon will appear on your desktop.

To capture an image of the *entire screen*, use *Apple-shift-3*.

11.5.3. How to get a screen shot under Linux

Gimp, a public-domain package similar to Photoshop, makes it easy to get a screen shot.

1. Bring up Gimp. From the *Start* menu, select *Graphics*, then *GIMP Image Editor*.
2. From the Gimp menu, select *File*, then *Acquire*, then *Screen Shot...*
3. To capture an image of one window, select the radiobutton for *a Single Window*, increase the delay time under *Select Window After...Seconds Delay*, and click *OK*. After the delay you have specified, the cursor will turn into a crosshair symbol. Move the cursor into the desired window and click the left mouse button.

To capture an image of the entire screen, select the radiobutton for *the Whole Screen*, increase the delay time under *Grab After ... Seconds Delay*, and click *OK*. After the delay you have specified, Gimp will take a snapshot of the whole screen.

4. A window containing your snapshot will appear on your screen. You can use Gimp to crop the image, adjust color and contrast, and the other usual photo processing options. Most operations start by right-clicking on the image.
5. To save the image to a file, move the mouse into the image window and right-click. Then select *File*, then *Save*.

You can navigate to any directory by clicking in the *Folder* window. Click on *.. /* to go up one directory; to move down into a directory, click on that directory's name in the *Folder* window.

Once you are in the right directory, type the name you want to give the graphics file in the *Selection* field, including the file type suffix such as *.png* or *.jpg*, and press *Enter*.

12. Tables

DocBook has extensive features that help you present data in a tabular form. Here is an example of a small table:

Table 1. All-time NBA free throw percentages

Player	FTA	FTM	Pct.
Rick Barry	4,243	3,818	.900
Calvin Murphy	3,864	3,445	.892

Here is how the XML source for the table looks:

```
<table frame="topbot" pgwide="0">
  <title>All-time NBA free throw percentages</title>
  <tgroup cols="4">
    <colspec colwidth="3*" align="left" colnum="1" colname="player"/>
    <colspec colwidth="*" align="right" colnum="2" colname="fta"/>
    <colspec colwidth="*" align="right" colnum="3" colname="ftm"/>
```

```

<colspec colwidth="*" align="right" colnum="4" colname="pct"/>
<thead>
  <row rowsep="1">
    <entry>Player</entry>
    <entry>FTA</entry>
    <entry>FTM</entry>
    <entry>Pct.</entry>
  </row>
</thead>
<tbody>
  <row>
    <entry>Rick Barry</entry>
    <entry>4,243</entry>
    <entry>3,818</entry>
    <entry>.900</entry>
  </row>
  <row>
    <entry>Calvin Murphy</entry>
    <entry>3,864</entry>
    <entry>3,445</entry>
    <entry>.892</entry>
  </row>
</tbody>
</tgroup>
</table>

```

Let's look at these tags and their attributes in detail.

- The **table** element encloses the entire table. The attribute `frame="topbot"` specifies that ruled lines be placed over the top and bottom of the table as a whole. The attribute `pgwide="0"` tells DocBook to place the table within the current paragraph width.

If a page break falls in the middle of a table, the heading will be repeated on the new page.

- Every table must be titled, so the next thing after the **table** opening tag must be a **title** element.
- The **tgroup** element encloses the entire rest of the table. Its `cols` attribute specifies the number of columns for the table as a whole.
- Specify the presentation of each column in the table with a **colspec** element.

The `colwidth` attribute specifies the width of the column. In this example, we want the first column to be three times as wide as the other columns, so we use a value of `colwidth="3*"` for the first column and `colwidth="*"` for the rest.

The `align` attribute specifies whether the contents of each column are to be positioned to the left or right side.

- The **thead** element encloses the heading section of the table. The heading consists of a **row** element, containing one **entry** element for each heading.

Note that the **row** element has an attribute `rowsep="1"` that places a ruled line after that row. The rows in the table body do not have that attribute and are not separated by rules.

- The **tbody** element comes after the last **colspec** element, and encloses the actual body of the table.
- Each row of the table is enclosed in a **row** element.
- Each cell in the table is enclosed within an **entry** element.

The sections below discuss a number of ways you can control the appearance of your table, but not all details of table construction are covered. For the full gory details, see the *CALS table model Document Type Definition*¹⁹.

12.1. Ruled lines in tables

The default table appearance is to have *rules* (ruled lines) around the outside of the table and between all rows and columns.

You can control where rules appear by adding attributes to the various elements of your table. There is a hierarchy of elements:

1. Attributes of the `table` element specify values for the table as a whole.
2. Attributes of `colspec` specify values for all entries in a column, and may override values set in the `table` element.
3. Attributes of the `row` element can override higher-level values.
4. Attributes of the `entry` element can in turn override all higher-level values.

The example table above shows this process. The `frame="topbot"` attribute of the `table` element specifies that rules appear only above and below the table as a whole. But the `rowsep="1"` attribute of the first `row` element overrides that specification, forcing a rule to appear below the first row.

Here are the elements and attributes that affect ruled lines:

- In the `table` element, the `frame` attribute can have any of these values:

Value	Meaning
<code>all</code>	Rules are placed above and below the table, on the left and right sides, between columns, and between rows.
<code>none</code>	No rules are used in the table.
<code>sides</code>	Rules are placed only at the left and right sides of the table.
<code>top</code>	A rule is placed only above the table.
<code>bottom</code>	A rule is placed only below the table.
<code>topbot</code>	Rules are placed above and below the table.

- Any of the elements `table`, `tgroup`, `colspec`, and `entry` can have a `colsep` attribute.

Use `colsep=0` to suppress rules to the right of a column, or `colsep="1"` to place rules to the right of a column.

The attribute for the `table` element sets the default for the table as a whole; the attribute for `tgroup` overrides the value for `table`; and so forth, with the value for deeper elements overriding the values of all shallower elements.

- Any of the elements `table`, `tgroup`, `row`, and `entry` can have a `rowsep` attribute.

Use `rowsep=0` to suppress the rule below the row (or cell, for the `entry` element). Use `rowsep="1"` to place a rule below the row (or cell).

As with the `colsep` attribute, values of this attribute for deeper elements override values in shallower elements.

¹⁹ <http://www.oasis-open.org/cover/tr9502.html>

12.2. Controlling table dimensions

Normally, your table will be fit into the page width remaining after the current indentation level is subtracted from the total page width. However, if you need more width, use a `pgwide="1"` attribute in the `table` element. This will give you the whole width of the page to work with.

Your other job is to distribute this width among the columns of the table. You can do this in two ways:

- You can assign a specific width to each column of the table. However, this can make life more difficult for people reading the Web version of your document if their window is too narrow for your table.
- A better way is to specify the relative width of the columns. The advantage of this method is that in Web form your table can be resized to conform to the width of the browser window.

Regardless of which method you use, the width of each column is specified in the `colwidth` attribute of the `colspec` element, which can take these values:

- A number followed by a unit of measure. Units include: `pt` (printer's points, about 1/72 inch), `pi` (picas, about 1/6 inch), `mm` (millimeters), `cm` (centimeters), or `in` (inches). For example, `colwidth="5.5cm"` would specify a width of 5.5 centimeters.
- A number followed by an asterisk. This allows you to specify relative column widths. With this method, all the numbers are added up, and the space is divided according to the coefficients. The numbers can be integers or fixed-point constants such as `5.25`.

For example, suppose your table has four columns and the values of the `colwidth` attribute are `"3*", "*","2*", "2"`. The sum of these coefficients (the second value is the same as `"1"`) is 8. The resulting table would allocate 3/8 of the width to the first column, 1/8 of the width to the second column, and 1/4 of the width to each of the third and fourth columns.

12.3. Controlling alignment in tables

By *alignment*, we mean the positioning of elements within the cells of a table. Since most cells will not be filled exactly, we need ways of controlling where the content is placed within the cell, both horizontally and vertically.

Horizontal alignment is controlled by the `align` attribute. This attribute can be used:

- in a `tgroup` element, to set the default alignment for the table as a whole;
- in a `colspec` element, to set the default alignment for one column; or
- in an `entry` element, to set the alignment for a specific cell.

Attribute values for deeper elements override those for shallower elements.

The value of the `align` attribute can be any of the following:

Value	Meaning
<code>left</code>	Content is aligned to the left side of each cell.
<code>right</code>	Content is aligned to the right side.
<code>center</code>	Content is centered.
<code>justify</code>	Text is shown as a justified paragraph, stretched to the full width of the cell.

Vertical alignment is controlled by the `valign` attribute. This attribute can be used:

- in a `row` element, to set the default alignment for that row; or
- in an `entry` element, to change the alignment of a single cell.

The value of the `valign` attribute can be any of these:

Value	Meaning
top	Content is aligned to the top side of each cell.
middle	Content is centered vertically in the cell.
bottom	Content is aligned to the bottom of the cell.

12.4. Horizontal (column) spanning in tables

So far we have talked only about tables with a cell at the intersection of each row and column. In the real world, though, we often need to *span* cells, that is, to combine two or more cells into a larger area that holds a single item.

Here is the procedure for column spanning, where multiple cells of the same row are combined.

1. To each `colspec` element, add a `colname` attribute that specifies a unique name for that column.
2. Inside the `entry` element to be spanned, add two attributes. Set the `namest` attribute value to the name of the column where the spanning is to start. Set the `nameend` attribute value to the name of the ending column.

Here is a small table that demonstrates spanning. Note that the last two column headings are each centered over two columns:

Table 2. Rising and setting of Venus, 1994

		20° N. Lat.		30° N. Lat.	
		Rise	Set	Rise	Set
Jan.	1	6:21	17:14	6:43	16:52
	11	6:35	17:31	6:56	17:10

Here's the source for the first part of this table:

```
<table id='venus-table'>
  <title>Rising and setting of Venus, 1994</title>
  <tgroup cols="6" align="right">
    <colspec align="left" colname="month"/>
    <colspec colwidth="4em" colname="day"/>
    <colspec colwidth="3em" colname="rise-20"/>
    <colspec colwidth="5em" colname="set-20"/>
    <colspec colwidth="5em" colname="rise-30"/>
    <colspec colwidth="5em" colname="set-30"/>
  <thead>
    <row>
      <entry namest="month" nameend="day"/>
      <entry namest="rise-20" nameend="set-20" align="center">
        20° N. Lat.
      </entry>
      <entry namest="rise-30" nameend="set-30" align="center">
        30° N. Lat.
      </entry>
    </row>
  </thead>
</table>
```

```

<row>
  <entry namest="month" nameend="day"/>
  <entry>Rise</entry>
  <entry>Set</entry>
  <entry>Rise</entry>
  <entry>Set</entry>
</row>
</thead>
<tbody>
  <row>
    <entry>Jan.</entry>
    <entry>1</entry>
    <entry>6:21</entry>
    <entry>17:14</entry>
    <entry>6:43</entry>
    <entry>16:52</entry>
  </row>
  ...

```

12.5. Vertical (row) spanning in tables

You can also make a cell span multiple rows vertically. To do this:

1. Inside the cell's `entry` element, add an attribute `morerows="N"`, where *N* is the number of *additional* rows to be spanned. For example, an attribute of `morerows="2"` would create a cell that spans *three* rows.
2. Wherever a cell is vertically spanned, omit the `entry` elements from the rows into which it is spanned. For example, if the cell in column 1 of row 1 spans three rows, omit the `entry` element for column 1 in rows 2 and 3.

Here is our Venus table rearranged to demonstrate vertical spanning:

Table 3. Rising and setting of Venus, 1994

Month	Day	20° N. Lat.		30° N. Lat.	
		Rise	Set	Rise	Set
Jan.	1	6:21	17:14	6:43	16:52
	11	6:35	17:31	6:56	17:10

Here is the `thead` section of the modified table; the rest of the table is as described in Section 12.4, "Horizontal (column) spanning in tables" (p. 29).

```

<thead>
  <row>
    <entry morerows="1" valign="bottom">Month</entry>
    <entry morerows="1" valign="bottom">Day</entry>
    <entry namest="rise-20" nameend="set-20" align="center">
      20° N. Lat.
    </entry>
    <entry namest="rise-30" nameend="set-30" align="center">
      30° N. Lat.

```

```

        </entry>
    </row>
    <row>
        <entry>Rise</entry>
        <entry>Set</entry>
        <entry>Rise</entry>
        <entry>Set</entry>
    </row>
</thead>

```

Although the table has six columns, the second row in the `thead` element has only the four cells for columns 3-6, because columns 1 and 2 in those rows are occupied by the vertically spanned cells from the previous row.

You can combine vertical and horizontal spanning. If you do, the spanned cell will always occupy a rectangular block of the table. For example, if the cell in row 15, column 3, spans four columns and two rows, it will occupy columns 3-6 of rows 15-16.

13. Including *TeX* and *LaTeX* math

You can include mathematical formulae written in *TeX* and *LaTeX* in your document. Here is an example:

$$(14) \quad \int \tanh^{-1} \frac{x}{a} dx = x \tanh^{-1} \frac{x}{a} + \frac{a}{2} \log(a^2 - x^2), \quad \left(\left| \frac{x}{a} \right| < 1 \right)$$

There are some limitations:

- The technique works best for equations displayed as a block element, interrupting any text paragraph.
- Each displayed equation must be placed in a separate file. For hints on writing these files, see Section 13.1, “Preparing a formula with *LaTeX*” (p. 31) or Section 13.2, “Preparing a formula with *TeX*” (p. 32).
- Displayed equations are set flush to the left margin. The best workaround for this is to number your equations on the left; this will center the math part of the line.
- You can include bits of math as inline elements within a paragraph; see Section 13.5, “Simple inline math” (p. 35) and Section 13.6, “Inline math using *LaTeX* or *TeX*” (p. 36).
- The procedure is somewhat involved; see Section 13.3, “Processing your math files for inclusion” (p. 32). However, adding a few rules to your `Makefile` automates the entire procedure: see Section 13.4, “Automating math display production with your `Makefile`” (p. 34).

13.1. Preparing a formula with *LaTeX*

To use a *LaTeX* displayed formula, the formula must reside in a separate file, and you must follow a specific structure. Here is the *LaTeX* source file for the example in Section 13, “Including *TeX* and *LaTeX* math” (p. 31).

```

% lamath.tex: Sample of LaTeX math for inclusion in DocBook
%
\documentclass[leqno]{article}
\pagestyle{empty}
\setlength{\textwidth}{6in}

```

```

\begin{document}
\setcounter{equation}{13}
\begin{equation}
\int \tanh^{-1}\{x\over a\}dx =
x \tanh^{-1}\{x\over a\} + \{a\over 2\}\log(a^2-x^2), \quad
\left(\left| x\over a \right| < 1\right)
\end{equation}
\end{document}

```

- The option [leqno] instructs *LaTeX* to place equation numbers on the left side.
- The conversion process selects everything on the page and puts it into a rectangular box. Hence, a page number would force the box to be page-sized. The `\pagestyle{empty}` command suppresses page numbering.
- The line `\setlength{\textwidth}{6in}` sets the width of the text column to six inches, which matches the text column width in the PDF output from DocBook.
- Use a line `\setcounter{equation}{N}` to set the equation number to one less than the desired equation number. A value of 13 here will number the equation as (14).
- Place the equation in a `\begin{equation}...\end{equation}` environment so that the equation will be numbered.

The math itself is expressed using the usual *LaTeX* conventions.

13.2. Preparing a formula with TeX

If you prefer to use Plain *TeX*, here is the same example in that notation.

```

% math.tex: Sample of TeX math for inclusion in DocBook
%
\hsize=6in
\nopagenumbers
$$\int \tanh^{-1}\{x\over a\}dx =
x \tanh^{-1}\{x\over a\} + \{a\over 2\}\log(a^2-x^2), \quad
\left(\left| x\over a \right| < 1\right)
\leqno{(14)}
$$
\bye

```

- The line `\hsize=6in` sets the text column width to six inches.
- Because the production process wraps a box around whatever text is on the page, use `\nopagenumbers` to turn off page numbering.
- To place equation numbers on the left, use `\leqno` or `\leqalignno`.

13.3. Processing your math files for inclusion

Once you have written out your formula in *LaTeX* or *TeX* format, follow this procedure to prepare artwork for inclusion in DocBook. If you use a *Makefile*, you can automate this procedure; see Section 13.4, “Automating math display production with your *Makefile*” (p. 34).

1. Translate the file to DVI format. For *LaTeX*, the command is:

```
latex file.tex
```

For *TeX*:

```
tex file.tex
```

For example, if your source file is `eq14.tex`, this will create a file named `eq14.dvi`.

2. If there are any errors, fix them and recompile. Use *x_dvi* or another DVI viewer to inspect the typeset output and fix any content or appearance problems.
3. Convert the `.dvi` file to Encapsulated Postscript. To continue the example:

```
dvips -E eq14.dvi -o eq14.eps
```

The `-E` option forces *dvips* to produce EPS output, shrinking the bounding box to include only the marks on the page.

4. Convert the EPS file to Encapsulated PDF. This command will build a file named `eq14.pdf` that can be included in DocBook PDF output:

```
epstopdf eq14.eps
```

5. Convert the Encapsulated PDF file to a grayscale image in PGM format. The *pdftoppm* utility does this conversion. Its `-gray` option uses anti-aliasing to produce an image that is considerably better-looking than a pure black and white image. The first argument is the input file. The second argument is the “base name”; each page of the input file is written to a file whose name is that base name with a suffix of the form `-NNNNNN.pgm`, where `NNNNNN` is the page number. In our case, there will be only one page.

The *pdftoppm* program also accepts a resolution in dots per inch, but the default (`-r 150`) is reasonable.

To continue the example, this would build file `eq14-1.pgm`:

```
pdftoppm -gray eq14.pdf eq14
```

6. Finally, convert the `.pgm` file to JPEG format using *pnmtojpeg*. In our example:

```
pnmtojpeg eq14-1.pgm >eq14.jpg
```

7. In your DocBook source file, encode the equation like this:

```
<informalequation>
  <mediaobject>
    <imageobject role="html">
      <imagedata fileref="file.jpg"/>
    </imageobject>
    <imageobject role="fo">
      <imagedata fileref="file.pdf"/>
    </imageobject>
  </mediaobject>
</informalequation>
```

The `informalequation` element is preferable to the `equation` element, since it does not number the equation or give it a title.

13.4. Automating math display production with your Makefile

If you are using a `Makefile` as described in Section 4.2, “The DocBook workflow cycle” (p. 5), you can add rules to it that automate the procedure described in Section 13.3, “Processing your math files for inclusion” (p. 32).

Here is an annotated `Makefile` for this example file, illustrating how to automate the procedures for converting `TEX` equations to the forms needed by DocBook.

First, we define a few variables. `TARGET` is the base name of the document and its product files. `SOURCE` is the document's XML source file. In this example, the DocBook source file is `example.xml`.

```
TARGET      = example
SOURCE      = $(TARGET).xml
```

The next lines are a part of the stock TCC DocBook `Makefile` template, defining the path to the style sheets, and options for the HTML and PDF conversion routes.

```
TCC_XSL      = /u/www/docs/tcc/doc/docbook43
XSLT_HTML_OPT =
XSLT_FO_OPT  =
XEP_OPT      = -quiet
```

The variable `EQUATIONS` enumerates the individual `TEX` source files for each displayed equation. The variables `PDF_EQUATIONS` and `JPG_EQUATIONS` are the names of the product files required by DocBook for print and Web, respectively. In this example, there are three equations, residing in files `eq1.tex`, `eq2.tex`, and `eq3.tex`.

```
EQUATIONS    = eq1.tex eq2.tex eq3.tex
PDF_EQUATIONS = ${EQUATIONS:.tex=.pdf}
JPG_EQUATIONS = ${EQUATIONS:.tex=.jpg}
```

The `make` application won't work properly unless the `.SUFFIXES` variable is set to a list of all the file name suffixes used in its rules.

```
.SUFFIXES: .xml .html .fo .ps .eps .pdf .tex .dvi .pgm .jpg
```

The next section is the suffix rules used to convert one file format into another. The first three are the normal rules for building HTML and PDF, from the TCC's stock DocBook `Makefile`.

```
.xml.html:
    xsltproc -o $@ $(XSLT_HTML_OPT) $(TCC_XSL)/tcc_html.xsl $<

.xml.fo:
    xsltproc -o $@ $(XSLT_FO_OPT) $(TCC_XSL)/tcc_fo.xsl $<

.fo.pdf:
    xep $(XEP_OPT) $< $@
```

The next sequence of rules exactly reflects the steps described in the procedure above for converting the *LaTeX* source files to PDF and JPG form.

```
.tex.dvi:
    latex $<

.dvi.eps:
```

```

dvips -E $< -o $*.eps

.eps.pdf:
    epstopdf $<

.pdf.pgm:
    pdftoppm -gray $< $*; \
    mv $*-1.pgm $*.pgm

.pgm.jpg:
    pnmtojpeg $< >$@

```

If you are using *TeX* instead of *LaTeX*, change the first rule above to:

```

.tex.dvi:
    tex $<

```

The first, and therefore default, *make* target is `all`. It depends on `web`, which builds the Web side, and `pdf`, which builds the PDF. These first rules are from the stock `Makefile`, except that we note that the HTML route depends on the JPG forms of the equations, and the PDF route depends on the Encapsulated PDF equations.

```

all: web pdf

web: $(TARGET).html

$(TARGET).html: $(SOURCE) $(JPG_EQUATIONS)

pdf: $(TARGET).pdf $(PDF_EQUATIONS)

$(TARGET).pdf: $(SOURCE) $(PDF_EQUATIONS)

```

Finally, the `clean` target removes the HTML and PDF product files, as well as the included equations.

```

clean:
    rm -f $(TARGET).fo $(TARGET).pdf *.html $(JPG_EQUATIONS)
    $(PDF_EQUATIONS)

```

13.5. Simple inline math

There are two ways to include mathematical formulae as inlines within a paragraph.

If your expressions are relatively simple, you can enclose them inside an element like this:

```
<phrase role="math">...</phrase>
```

This will show all letters in italics. Here is an example:

We must prove that $a^2 < b_x$ for all cases.

Here is the source for that example:

```

We must prove that
<phrase role='math' >a<superscript >2</superscript

```

```
>&lt;b<subscript >x</subscript ></phrase >  
for all cases.
```

13.6. Inline math using *LaTeX* or *TeX*

You can include *LaTeX* or *TeX* math formulae as inlines. The preparation is the same as for displayed math. However, the baseline of these elements might not line up with the baseline of the surrounding text. Because this technique is somewhat ugly, it should be used only when you need characters unavailable in regular DocBook (see Section 18, “Special characters” (p. 41)).

The graphic is included using the techniques described in Section 11.4, “Inline graphics” (p. 24).

Here is an example of this technique:

We must prove that $a^2 < b_x$ in all cases.

And here is the source:

```
<para>  
  We must prove that  
  <inlinemediaobject>  
    <imageobject role='html'>  
      <imagedata fileref='inmath.jpg' />  
    </imageobject>  
    <imageobject role='fo'>  
      <imagedata fileref='inmath.pdf' />  
    </imageobject>  
  </inlinemediaobject>  
  in all cases.  
</para>
```

14. User-defined entities

You may wish to abbreviate a frequently-used word or phrase as an *entity* in your document. This allows you to substitute a short string of the form $\&n$; wherever that word or phrase is used, and the full text will be substituted automatically.

The entity's name part n is a symbolic name following the usual XML conventions (starting with a letter, and containing only letters, digits, underbars “_”, and hyphens “-”).

Place your entity definitions in the $\<!DOCTYPE>$ declaration at the top of your document, enclosed in square brackets and just before the closing $\>$. Each declaration looks like this:

```
<!ENTITY n "T">
```

where n is the entity's name and T is the replacement text.

For example, suppose you are developing a product under the internal code name *DaisyMatic*, and you want to write the manual without having to know the final, public name of the product would be. You can define an entity $\&product$; as the text “*DaisyMatic*” by changing your document type declaration to look like this:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"  
  "http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd"
```

```
[ <!ENTITY product "DaisyMatic">
]
>
```

With the above definition, anyplace that `&product;` appears in your XML source file, it will be replaced by “DaisyMatic”.

Then, when the marketing department decides that the external product name is going to be “Mega-MonsterMatic-3000”, just change the replacement text and rebuild your document, and the new product name will appear everywhere:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd"
[ <!ENTITY product "MegaMonsterMatic-3000">
]
>
```

You can have any number of entity declarations between the square brackets “[. . .]”.

A highly useful technique is to declare entities whose values include other entities. For example, the URL of the current document is:

```
http://www.nmt.edu/tcc/help/pubs/docbook43/
```

We would like it to be easy to move all references to this page and its sub-pages. Furthermore, we would like it to be easy to fix references to other documents elsewhere in the Tech Computer Center structure that may someday move to a different URL.

Consequently, the source for the current document (see Section 2, “Relevant online files” (p. 3)) has these entities in its DOCTYPE declaration:

```
<!ENTITY selfName "docbook43">
<!ENTITY nmtURL "http://www.nmt.edu/">
<!ENTITY tccURL "&nmtURL;tcc/">
<!ENTITY helpURL "&tccURL;help/">
<!ENTITY pubsURL "&helpURL;pubs/">
<!ENTITY selfURL "&pubsURL;&selfName;/">
<!ENTITY selfPDFName "&selfName;.pdf">
<!ENTITY selfPDFFile "<filename>&selfPDFName;</filename>">
<!ENTITY selfXMLName "&selfName;.xml">
<!ENTITY selfXMLFile "<filename>&selfXMLName;</filename>">
```

Here is a table showing how these are expanded:

<code>&selfName;</code>	docbook43
<code>&nmtURL;</code>	http://www.nmt.edu/
<code>&tccURL;</code>	http://www.nmt.edu/tcc/
<code>&helpURL;</code>	http://www.nmt.edu/tcc/help/
<code>&pubsURL;</code>	http://www.nmt.edu/tcc/help/pubs/
<code>&selfURL;</code>	http://www.nmt.edu/tcc/help/pubs/docbook43/
<code>&selfPDFName;</code>	docbook43.pdf
<code>&selfPDFFile;</code>	<filename>docbook43.pdf</filename>
<code>&selfXMLName;</code>	docbook43.xml

<code>&selfXMLFile;</code>	<code><filename>docbook43.xml</filename></code>
<code>&selfURL;&selfPDFName;</code>	<code>http://www.nmt.edu/tcc/help/pubs/docbook43/doc-book43.pdf</code>

15. Breaking your document into multiple files

For larger documents, it is often convenient to break the document into more than one file, so you can work on a specific chapter or section by itself.

This is easy to do because of another type of entity declaration that you can put inside your DOCTYPE. Here's the general form:

```
<!ENTITY new-name SYSTEM "filename">
```

This defines a new entity named “*&new-name;*”. If this entity appears in your document, the effect is to insert the contents of file *filename* at that point.

Here's an example. Suppose you want to break your document up into four files—a top-level file named `mydoc.xml` and three subsidiary files named `head.xml`, `body.xml`, and `tergum.xml`. File `mydoc.xml` might look like this:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.3//EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd"
[
  <!ENTITY head SYSTEM "head.xml">
  <!ENTITY body SYSTEM "body.xml">
  <!ENTITY tail SYSTEM "tergum.xml">
]
>
<article>
  <articleinfo>
    ... <!-- Usual article info content here -->
  </articleinfo>
  &head;
  &body;
  &tail;
</article>
```

There is one drawback to this method. If you are using special character entities such as `°` (the degree symbol, °), your HTML and PDF output files will still build normally, but some editing tools (such as *emacs nxml-mode*²⁰) will no longer validate these character entities, because they see only the current file, and in the subsidiary files there is no `<!DOCTYPE>` declaration to tell them where the entities are defined.

The workaround for this problem is to use the alternate form for each entity that uses the hexadecimal Unicode character value. For example, the degree symbol entity “`°`” can also be expressed as “`°`”. For a complete list of all special character entities in both name and numeric form, see Section 18, “Special characters” (p. 41).

²⁰ <http://www.nmt.edu/tcc/help/pubs/nxml/>

16. Decluttering the project directory

Normally, the generated `.html` pages are placed in the same directory as the source `.xml` file. For larger documents with many sections, this can lead to a directory with a lot of files in it.

Follow this procedure to relocate all the generated `.html` files to a `web/` subdirectory.

1. You will need to modify your `Makefile`. See Section 19.3, “`make - large: A Makefile for large projects`” (p. 51) for a discussion of the changed rules.
2. You will need to create a file named `.htaccess` in the same directory as your `.xml` source file. This file instructs Web browsers to redirect all requests for Web files in this directory to look in the `web/` subdirectory instead.

Here is a generic `.htaccess` file:

```
RedirectMatch P/([^\./]*)\.html) P/web/$1 [R=301]
RedirectMatch P/web/homepage.html P/web/index.html
```

Wherever `P` appears in the above skeleton, substitute the URL of the document's directory, starting after the `http://server.domain` part, but including the initial `"/`. The `"[R=301]"` informs browsers that the page has moved permanently; without this, the *Back* button would not work correctly.

For example, here is the `.htaccess` file for this document:

```
RedirectMatch /tcc/help/pubs/docbook43/([^\./]*)\.html) \
/tcc/help/pubs/docbook43/web/$1 [R=301]
RedirectMatch /tcc/help/pubs/docbook43/web/homepage.html \
/tcc/help/pubs/docbook43/web/index.html
```

17. Literate programming with DocBook

Literate programming²¹ is a venerable technique invented in the early 1980s by Don Knuth.

There are four approaches to documenting programs:

- No documentation at all. Common, but highly unprofessional.
- Documentation separate from the program's source code.
- Documentation embedded within the program's source code. Systems such as `perldoc`²² and `pydoc`²³ allow the programmer to use specially formatted comments to document the code.

Although almost any documentation is better than no documentation, this approach limits the toolkit of the documentation writer rather severely, bereft of tools for such features as figures and serious tables.

- With literate programming, the *program source code is inside the document*, embedded within a narrative that explains the author's thought processes. This gives you the full toolset of your documentation system, and allows you to use pictures, diagrams and tables to explain your design.

²¹ http://en.wikipedia.org/wiki/Literate_programming

²² <http://en.wikipedia.org/wiki/Perldoc>

²³ <http://en.wikipedia.org/wiki/Pydoc>

Lightweight literate programming²⁴ is a simplification of Knuth's approach: instead of Knuth's *cweb* tool, you write your documentation and source code using a regular word processing system.

If you use DocBook as described in this document, you can easily embed source code for later extraction and use. Encode each fragment in a `programlisting` element with this general form:

```
<programlisting role="outFile:F">
    your source code here
</programlisting
```

where *F* is the name of the file to which you want this fragment written.

Here is an example. Suppose you are writing a C program that consists of two files, a header file named `foo.h` and a source code file named `foo.c`. Part of your XML documentation for this program might look like this:

```
<programlisting role='outFile:foo.h'>
    stuff to be written to foo.h
</programlisting>
...
<programlisting role='outFile:foo.c'>
    stuff to be written to foo.c
</programlisting>
```

If you name the same output file in more than one `programlisting` section, all the fragments will appear in the output file in the same order in which they occur in your document.

Once you have encoded your program in this way, you will build the HTML and PDF renderings in the usual way. To extract the code from the DocBook XML file, see *litxml: A source extractor for lightweight literate programming*²⁵.

If you are using the Unix *make* application to build files in your document, see Section 19.2, “*make-lit: A Makefile for literate programming*” (p. 48).

17.1. Controlling spurious blanks and blank lines

Every character after the opening `<programlisting>` tag, up to the closing `</programlisting>` tag, is written to the file. Hence, if the opening tag is on the line before the first line of your program fragment, the content written to the output file will start with an empty line.

Also, if your closing `</programlisting>` tag is indented, the output will include the indentation whitespace.

To avoid these extra spaces, move the closing “`>`” of the opening tag to just before the first line of your fragment, and place the closing tag in column 1 of your DocBook source file.

For example, here is how a two-line `bash` script must be encoded so that only these two lines will appear in the output. Suppose your XML is currently indenting four spaces in this section.

```
<para>
    Here is the <code>whereami</code> script:
</para>
<programlisting role='outFile:whereami'
>#!/bin/bash
```

²⁴ <http://www.nmt.edu/~shipman/soft/litprog/>

²⁵ <http://www.nmt.edu/tcc/help/lang/python/examples/litxml>

```

echo $HOSTNAME
</programlisting>
<para>
  etc.
</para>
...

```

18. Special characters

A wide variety of special symbols are available in DocBook. These *character entities* always start with an ampersand (&) and end with a semicolon (;).

For more information on named Unicode characters, see *XML entity definitions for characters*²⁶. However, there is no guarantee that any of these entities will work in the local toolchain; it is best to test any characters before using them in publications.

18.1. Universally available entities

Entities for these five special characters are always available:

<	<
>	>
&	&
'	'
"	"

18.2. International character entities

DocBook supports a modest set of letters with diacritical marks. Use the *entity name* shown in the first column to get the character shown in the second column. The third column shows an alternative form using a numeric value; such entities are less user-friendly but may be necessary in some circumstances.

Entity	Character	Equivalent
á	á	á
Á	Á	Á
â	â	â
Â	Â	Â
à	à	à
À	À	À
å	å	å
Å	Å	Å
ã	ã	ã
Ã	Ã	Ã

²⁶ <http://www.w3.org/TR/xml-entity-names/>

Entity	Character	Equivalent
ä	ä	ä
Ä	Ä	Ä
æ	æ	æ
Æ	Æ	Æ
ç	ç	ç
Ç	Ç	Ç
ð	ð	ð
Ð	É	é
é	é	é
É	É	É
ê	ê	ê
Ê	Ê	Ê
è	è	è
È	È	È
&euuml;	ë	ë
&Euuml;	Ë	Ë
í	í	í
Í	Í	Í
î	î	î
Î	Î	Î
ì	ì	ì
Ì	Ì	Ì
ï	ï	ï
Ï	Ï	Ï
ñ	ñ	ñ
Ñ	Ñ	Ñ
ó	ó	ó
Ó	Ó	Ó
ô	ô	ô
Ô	Ô	Ô
ò	ò	ò
Ò	Ò	Ò
ø	ø	ø
Ø	Ø	Ø
õ	õ	õ
Õ	Õ	Õ
ö	ö	ö

Entity	Character	Equivalent
Ö	Ö	Ö
ß	ß	ß
þ	þ	þ
Þ	Ð	Þ
ú	ú	ú
Ú	Ú	Ú
û	û	û
Û	Û	Û
ù	ù	ù
Ù	Û	Ù
ü	ü	ü
Ü	Ü	Ü
ý	ý	ý
Ý	Ý	Ý
ÿ	ÿ	ÿ

18.3. The Greek alphabet

Entity	Character	Equivalent	Name
&Agr;	A	Α	Alpha, uppercase
&agr;	α	α	Alpha, lowercase
&Bgr;	B	Β	Beta, uppercase
&bgr;	β	β	Beta, lowercase
&Dgr;	Δ	Δ	Delta, uppercase
&dgr;	δ	δ	Delta, lowercase
&EEgr;	H	Η	Eta, uppercase
&eeegr;	η	η	Eta, lowercase
&Egr;	E	Ε	Epsilon, uppercase
&egr;	ε	ε	Epsilon, lowercase
&Ggr;	Γ	Γ	Gamma, uppercase
&ggr;	γ	γ	Gamma, lowercase
&Igr;	I	Ι	Iota, uppercase
&igr;	ι	ι	Iota, lowercase
&Kgr;	K	Κ	Kappa, uppercase
&kgr;	κ	κ	Kappa, lowercase
&KHgr;	X	Χ	Chi, uppercase
&khgr;	χ	χ	Chi, lowercase
&Lgr;	Λ	Λ	Lambda, uppercase

Entity	Character	Equivalent	Name
&lgr;	λ	λ	Lambda, lowercase
&Mgr;	Μ	Μ	Mu, uppercase
&mgr;	μ	μ	Mu, lowercase
&Ngr;	Ν	Ν	Nu, uppercase
&ngr;	ν	ν	Nu, lowercase
&Ogr;	Ο	Ο	Omicron, uppercase
&ogr;	ο	ο	Omicron, lowercase
&OHgr;	Ω	Ω	Omega, uppercase
&ohgr;	ω	ω	Omega, lowercase
&Pgr;	Π	Π	Pi, uppercase
&pgr;	π	π	Pi, lowercase
&PHgr;	Φ	Φ	Phi, uppercase
&phgr;	φ	φ	Phi, lowercase
&PSgr;	Ψ	Ψ	Psi, uppercase
&psgr;	ψ	ψ	Psi, lowercase
&Rgr;	Ρ	Ρ	Rho, uppercase
&rgr;	ρ	ρ	Rho, lowercase
&Sgr;	Σ	Σ	Sigma, uppercase
&sgr;	σ	σ	Sigma, lowercase
&Tgr;	Τ	Τ	Tau, uppercase
&tgr;	τ	τ	Tau, lowercase
&THgr;	Θ	Θ	Theta, uppercase
&thgr;	θ	θ	Theta, lowercase
&Ugr;	Υ	Υ	Upsilon, uppercase
&ugr;	υ	υ	Upsilon, lowercase
&Xgr;	Ξ	Ξ	Xi, uppercase
&xgr;	ξ	ξ	Xi, lowercase
&Zgr;	Ζ	Ζ	Zeta, uppercase
&zgr;	ζ	ζ	Zeta, lowercase

18.4. Special symbols

Also supported are a wide variety of special symbols. Refer to O'Reilly's book *DocBook* for a complete list. Here is a selection:

Entity	Character	Equivalent	Name
∧	∧	∧	Logical and
≈	≈	≈	Approximately equal to
¦	∣	¦	Broken vertical bar

Entity	Character	Equivalent	Name
∩	\cap	∩	Set intersection
¢	¢	¢	Cents
©	©	©	Copyright
∪	\cup	∪	Set union
¤	¤	¤	Currency symbol
†	†	†	Dagger
‡	‡	‡	Double dagger
°	°	°	Degree symbol
÷	÷	÷	Division
↓	↓	↓	Down arrow
∅	∅	∅	Empty set
≡	≡	≡	Identical to
€	€	€	Euro currency symbol
∃	∃	∃	There exists
∀	∀	∀	For all
≥	≥	≥	Greater than or equal to
↔	↔	↔	Double-headed arrow
&Harr;	⇔	⇔	Double-headed arrow
…	...	…	Horizontal ellipsis
¡	¡	¡	Inverted exclamation point
¿	¿	¿	Inverted question mark
∈	∈	∈	Element of
«	«	«	Left double angle quotes
←	←	←	Left arrow
⇐	⇐	⇐	Left double arrow
“	“	“	Left double quote
„	„	„	Double low-nine quote
≤	≤	≤	Less than or equal to
‘	‘	‘	Left single quote
‚	,	‚	Single low-nine quote
—	—	—	Em-dash
·	·	·	Middle dot
−	−	−	Minus sign
 		 	Non-breaking space
–	–	–	En-dash
≠	≠	≠	Not equal to
∋	∋	∋	Contains

Entity	Character	Equivalent	Name
∉	∉	∉	Not an element of
∨	∨	∨	Logical or
¶	¶	¶	Paragraph (pilcrow) symbol
±	±	±	Plus or minus
£	£	£	Pounds sterling
′	'	′	Prime
″	''	″	Double prime
»	»	»	Right double angle quotes
→	→	→	Right arrow
⇒	⇒	⇒	Double right arrow
”	”	”	Right double quote
®	®	®	Registered
”	”	”	Double high reversed quote
’	'	’	Right single quote
§	§	§	Section symbol
∼	~	∼	Similar to
⊂	⊂	⊂	Subset of
⊆	⊆	⊆	Subset of or equal to
⊃	⊃	⊃	Superset of
⊇	⊇	⊇	Superset of or equal to
 		 	Thin space
×	×	×	Multiplication
™	™	™	Trademark
↑	↑	↑	Up arrow
¥	¥	¥	Yen symbol

19. Model files for make

In this section we present three model `Makefile` files that you can adapt for your DocBook project.

- Section 19.1, “make-basic: A basic `Makefile`” (p. 47): To get you started.
- Section 19.2, “make-lit: A `Makefile` for literate programming” (p. 48): When your DocBook document contains files to be extracted. See Section 17, “Literate programming with DocBook” (p. 39).
- Section 19.3, “make-large: A `Makefile` for large projects” (p. 51): To implement the techniques described in Section 16, “Decluttering the project directory” (p. 39).

19.1. make-basic: A basic Makefile

Here is file `make-basic`. Download this from Section 2, “Relevant online files” (p. 3), rename it as `Makefile`, and replace `your-project-name-here` with the name of your DocBook XML file, without its `.xml` suffix.

`make-basic`

```
# Makefile for DocBook projects: Basic version
# This file is extracted automatically from:
# http://www.nmt.edu/tcc/help/pubs/docbook43/
#=====
# Definitions
#-----
# BASENAME:      The base file name of the DocBook file.
# SOURCE:        The .xml file containing the document.
# WEB_TARGET:    Top-level generated HTML page name.
# FO_TARGET:     Intermediate file for generated PDF.
# PDF_TARGET:    Generated PDF file name.
# INDEX_BASE:    Base name for index of section id values.
# INDEX_FO:      Intermediate file for section index.
# INDEX_PDF:     Section index PDF file name.
# TCC_XSL:       Where the local customization layer lives.
# HTML_TRANSFORM: XSLT to build the HTML rendering.
# PDF_TRANSFORM: XSLT to build the PDF rendering.
# XSLT_HTML_OPT passes options to xsltproc to build the HTML form.
#   To push chunking down to subsections:
#     --stringparam chunk.section.depth 2
#   To move the first subsection to its own chunk:
#     --stringparam chunk.first.sections 1
#   To show subsections in the table of contents:
#     --stringparam toc.section.depth 3
# XSLT_FO_OPT: Options for hardcopy.
#   To show subsections in the table of contents:
#     --stringparam toc.section.depth 3
# XEP_OPT: Options to xep
#   To suppress most output:
#     -quiet
#-----
BASENAME      = your-project-name-here
SOURCE        = $(BASENAME).xml
WEB_TARGET    = index.html
FO_TARGET     = $(BASENAME).fo
PDF_TARGET    = $(BASENAME).pdf
INDEX_BASE    = toc
INDEX_FO      = $(INDEX_BASE).fo
INDEX_PDF     = $(INDEX_BASE).pdf
TCC_XSL       = /u/www/docs/tcc/doc/docbook43/
HTML_TRANSFORM = $(TCC_XSL)lit_html.xsl
FO_TRANSFORM  = $(TCC_XSL)lit_fo.xsl
XSLT_HTML_OPT = --stringparam chunk.section.depth 2 \
                --stringparam chunk.first.sections 1
XSLT_FO_OPT   =
XEP_OPT       = -quiet
```

```

#--
# Add the suffixes that we are interested in
#--
# .fo      XSL-FO output
# .html    HTML web pages
# .pdf     Page Description Format (Abode)
# .xml     Input files in DocBook XML format
#--
.SUFFIXES: .fo .html .pdf .xml

#=====
# Default suffix rules
#-----
.xml.html:
xsltproc -o @$ $(XSLT_HTML_OPT) $(HTML_TRANSFORM) $<

.xml.fo:
xsltproc -o @$ $(XSLT_FO_OPT) $(FO_TRANSFORM) $<

.fo.pdf:
xep $(XEP_OPT) $< @$

#=====
# Targets
#-----
all: web pdf index

web: $(WEB_TARGET)

$(WEB_TARGET): $(SOURCE)
xsltproc -o @$ $(XSLT_HTML_OPT) $(HTML_TRANSFORM) $<

pdf: $(PDF_TARGET)

$(PDF_TARGET): $(SOURCE)

index: $(INDEX_PDF)

$(INDEX_PDF): $(INDEX_FO)

$(INDEX_FO): $(SOURCE)
rm -f $(INDEX_FO); \
docbookindex $(SOURCE)

clean:
rm -f *.fo *.html *.pdf

```

19.2. make-lit: A Makefile for literate programming

This example `Makefile` implements the literate programming technique described in *Lightweight literate programming*²⁷. The basic idea is that rather than trying to embed documentation in your program, instead you embed the source code for your program within a DocBook document.

²⁷ <http://www.nmt.edu/~shipman/soft/litprog/>

You will need to install *litxml*, which is described (in literate form, of course) in *A source extractor for lightweight literate programming*²⁸.

The beginning section is similar to Section 19.1, “make-basic: A basic Makefile” (p. 47).

make-lit

```
# Makefile for DocBook projects: Literate version
# This file is extracted automatically from:
# http://www.nmt.edu/tcc/help/pubs/docbook43/
#=====
# Definitions
#-----
# BASENAME:      The base file name of the DocBook file.
# SOURCE:        The .xml file containing the document.
# WEB_TARGET:    Top-level generated HTML page name.
# FO_TARGET:     Intermediate file for generated PDF.
# PDF_TARGET:    Generated PDF file name.
# INDEX_BASE:    Base name for index of section id values.
# INDEX_FO:      Intermediate file for section index.
# INDEX_PDF:     Section index PDF file name.
# TCC_XSL:       Where the local customization layer lives.
# HTML_TRANSFORM: XSLT to build the HTML rendering.
# PDF_TRANSFORM: XSLT to build the PDF rendering.
# XSLT_HTML_OPT passes options to xsltproc to build the HTML form.
# To push chunking down to subsections:
#   --stringparam chunk.section.depth 2
# To move the first subsection to its own chunk:
#   --stringparam chunk.first.sections 1
# To show subsections in the table of contents:
#   --stringparam toc.section.depth 3
# XSLT_FO_OPT: Options for hardcopy.
# To show subsections in the table of contents:
#   --stringparam toc.section.depth 3
# XEP_OPT: Options to xep
# To suppress most output:
#   -quiet
#-----
BASENAME      = your-project-name-here
SOURCE        = $(BASENAME).xml
WEB_TARGET    = $(WEB_DIR)index.html
FO_TARGET     = $(BASENAME).fo
PDF_TARGET    = $(BASENAME).pdf
INDEX_BASE    = toc
INDEX_FO      = $(INDEX_BASE).fo
INDEX_PDF     = $(INDEX_BASE).pdf
TCC_XSL       = /u/www/docs/tcc/doc/docbook43/
HTML_TRANSFORM = $(TCC_XSL)lit_html.xsl
FO_TRANSFORM  = $(TCC_XSL)lit_fo.xsl
XSLT_HTML_OPT = --stringparam chunk.section.depth 2 \
                --stringparam chunk.first.sections 1
XSLT_FO_OPT   =
XEP_OPT       = -quiet
```

²⁸ <http://www.nmt.edu/tcc/help/lang/python/examples/litxml>

In the EXECUTABLES define, enumerate any files that are to be extracted and then made executable (we are assuming a Unix-based system here). Additionally, if there are any files that are to be extracted but do not need to be made executable, enumerate them in the MODULES define.

make-lit

```
#-----
# EXECUTABLES:  Files to be extracted and then made executable.
# MODULES:      Files to be extracted from the DocBook source.
#-----
MODULES          =
EXECUTABLES      =
CODE_TARGETS     = $(MODULES) $(EXECUTABLES)
```

The next section is the same as in make-basic.

make-lit

```
##-
# Add the suffixes that we are interested in
##-
# .fo          XSL-F0 output
# .html        HTML web pages
# .pdf         Page Description Format (Abode)
# .xml         Input files in DocBook XML format
##-
.SUFFIXES: .fo .html .pdf .xml

#=====
# Default suffix rules
#-----
.xml.html:
xsltproc -o $@ $(XSLT_HTML_OPT) $(HTML_TRANSFORM) $<

.xml.fo:
xsltproc -o $@ $(XSLT_F0_OPT) $(FO_TRANSFORM) $<

.fo.pdf:
xep $(XEP_OPT) $< $@
```

In the target section, we define a new target named code. The rule for this target runs *litlxml*. We also define a target named index that runs *docbookindex* to build a toc.pdf file enumerating all the section ID values.

make-lit

```
#=====
# Targets
#-----
all: web pdf index code

web: $(WEB_TARGET)

$(WEB_TARGET): $(SOURCE)
xsltproc -o $@ $(XSLT_HTML_OPT) $(HTML_TRANSFORM) $<

pdf: $(PDF_TARGET)

$(PDF_TARGET): $(SOURCE)
```

```

index: $(INDEX_PDF)

$(INDEX_PDF): $(INDEX_F0)

$(INDEX_F0): $(SOURCE)
    rm -f $(INDEX_F0); \
    docbookindex $(SOURCE)

code: $(CODE_TARGETS)

#--
# If there are any executables, change this rule to:
#   $(CODE_TARGETS): $(SOURCE)
#       litxml $<; \
#       chmod +x $(EXECUTABLES)
#--
$(CODE_TARGETS): $(SOURCE)
    litxml $<

clean:
    rm -f *.fo *.html *.pdf $(CODE_TARGETS)

```

19.3. make-large: A Makefile for large projects

This sample Makefile implements the technique described in Section 16, “Decluttering the project directory” (p. 39).

The beginning section is the same as in Section 19.1, “make-basic: A basic Makefile” (p. 47).

make-large

```

# Makefile for DocBook projects: Large-project version
# This file is extracted automatically from:
# http://www.nmt.edu/tcc/help/pubs/docbook43/
#=====
# Definitions
#-----
# BASENAME:      The base file name of the DocBook file.
# SOURCE:        The .xml file containing the document.
# WEB_DIR:       Subdirectory where .html files live.
# WEB_TARGET:    Top-level generated HTML page name.
# FO_TARGET:     Intermediate file for generated PDF.
# PDF_TARGET:    Generated PDF file name.
# INDEX_BASE:    Base name for index of section id values.
# INDEX_F0:      Intermediate file for section index.
# INDEX_PDF:     Section index PDF file name.
# TCC_XSL:       Where the local customization layer lives.
# HTML_TRANSFORM: XSLT to build the HTML rendering.
# PDF_TRANSFORM: XSLT to build the PDF rendering.
# XSLT_HTML_OPT passes options to xsltproc to build the HTML form.
# To push chunking down to subsections:
#   --stringparam chunk.section.depth 2
# To move the first subsection to its own chunk:

```

```

#      --stringparam chunk.first.sections 1
#      To show subsections in the table of contents:
#      --stringparam toc.section.depth 3
# XSLT_FO_OPT: Options for hardcopy.
#      To show subsections in the table of contents:
#      --stringparam toc.section.depth 3
# XEP_OPT: Options to xep
#      To suppress most output:
#      -quiet
#-----
BASENAME      = your-project-name-here
SOURCE        = $(BASENAME).xml

```

We add a new define named `WEB_DIR` that names the subdirectory where the generated HTML files will go. Regardless of the value of `BASENAME`, the style sheets build a root HTML file named `index.html`. By specifying that the `WEB_TARGET` is in a subdirectory, all the other files will be built in that subdirectory as well.

make-large

```

WEB_DIR      = web/
WEB_TARGET   = $(WEB_DIR)index.html
FO_TARGET    = $(BASENAME).fo
PDF_TARGET   = $(BASENAME).pdf
INDEX_BASE   = toc
INDEX_FO     = $(INDEX_BASE).fo
INDEX_PDF    = $(INDEX_BASE).pdf
TCC_XSL      = /u/www/docs/tcc/doc/docbook43/
HTML_TRANSFORM = $(TCC_XSL)lit_html.xsl
FO_TRANSFORM  = $(TCC_XSL)lit_fo.xsl
XSLT_HTML_OPT = --stringparam chunk.section.depth 2 \
                --stringparam chunk.first.sections 1
XSLT_FO_OPT   =
XEP_OPT       = -quiet

#--
# Add the suffixes that we are interested in
#--
# .fo      XSL-FO output
# .html    HTML web pages
# .pdf     Page Description Format (Abode)
# .xml     Input files in DocBook XML format
#--
.SUFFIXES: .fo .html .pdf .xml

#=====
# Default suffix rules
#-----
.xml.html:
xsltproc -o $@ $(XSLT_HTML_OPT) $(HTML_TRANSFORM) $<

.xml.fo:
xsltproc -o $@ $(XSLT_FO_OPT) $(FO_TRANSFORM) $<

.fo.pdf:

```

```
xep $(XEP_OPT) $< $@

#=====
# Targets
#-----
all: web pdf index

web: $(WEB_TARGET)
```

We can't use the default `.xml.html` rule above to build the HTML, because it assumes that the input `.xml` file and the output `index.html` file are in the same directory. Hence, we use an explicit build rule. Because the output file (`-o` option) to `xsltproc` is in a subdirectory, all the other files will be build there as well.

make-large

```
$(WEB_TARGET): $(SOURCE)
xsltproc -o $@ $(XSLT_HTML_OPT) $(HTML_TRANSFORM) $<

pdf: $(PDF_TARGET)

$(PDF_TARGET): $(SOURCE)

index: $(INDEX_PDF)

$(INDEX_PDF): $(INDEX_FO)

$(INDEX_FO): $(SOURCE)
rm -f $(INDEX_FO); \
docbookindex $(SOURCE)

clean:
rm -f *.fo *.html *.pdf
```

20. FOP: An older, free toolchain

This document assumes that you have XEP, a commercial product from RenderX²⁹. DocBook can also be processed using an open-source product, FOP from Apache³⁰.

The Tech Computer Center no longer supports the locally customized DocBook installation that works with FOP, but it is still available.

The software is generally reliable, but has the following specific limitations and problems.

20.1. FOP limitations

These features of XEP won't work in the FOP toolchain:

- You can't use any of the common entities such as `“`; for left double-quote.
- Graphics in PNG format are not supported.
- User-defined entities will not work.

²⁹ <http://www.renderx.com/>

³⁰ <http://xml.apache.org/fop/>

20.2. Bad page breaks

Sometimes a section or subsection title will appear at the bottom of a page, with the first line of the following paragraph at the top of the next page.

20.3. Using tables inside `<listitem>`

If you use a `<table>` or `<informaltable>` inside a `<listitem>` element (which in turn is part of a `<itemizedlist>` or `<variablelist>` construct), the Flow Object Processor (FOP) will loop endlessly.

20.4. Graphics file support

PNG (Portable Network Graphics) and EPS (Encapsulated PostScript) graphics files are not currently supported.

21. Converting DocBook-SGML 4.1 documents

If you have a document built under SGML-based DocBook, follow these steps to convert it to XML.

1. Replace the `<!DOCTYPE . . .>` declaration at the top of your document with the one from the first two lines of the `model` file in your working directory (assuming you followed the directory setup procedure above), or use `/u/www/docs/tcc/help/pubs/docbook43/user-kit/model`.
2. If you use pictures and other graphics, refer to the section above on including graphics. At this writing, only JPG and GIF formats are supported.

22. Converting DocBook 3.0 documents

If you have a document that worked under the old 3.0 version of DocBook, converting it is a simple matter:

- Move the `<title>` element from its old position inside the `<arheader>` element, to a position immediately after the `<article>` element at the very beginning of the document.
- Change the `<arheader>` tag and its closing `</arheader>` tag to `<articleinfo> ... </articleinfo>`.