AsciiDoc User Guide

Stuart Rackham <srackham@methods.co.nz>

Revision History

Revision 8.2.5	18 November 2007	SJR

Table of Contents

AsciiDoc is a text document format for writing short documents, articles, books and UNIX man pages. *AsciiDoc* files can be translated to HTML and DocBook markups using the `asciidoc(1)` command. *AsciiDoc* is highly configurable: both the *AsciiDoc* source file syntax and the backend output markups (which can be almost any type of SGML/XML markup) can be customized and extended by the user.

Introduction

This is an overly large document, it probably needs to be refactored into a Tutorial, FAQ, Quick Reference and Formal Reference.

If you're new to *AsciiDoc* read this section and the Getting Started section and take a look at the example *AsciiDoc* `*.txt` source files in the distribution `doc` directory.

Plain text is the most universal electronic document format, no matter what computing environment you use, you can always read and write plain text documentation. But for many applications plain text is not a viable presentation format. HTML, PDF and roff (roff is used for man pages) are the most widely used UNIX presentation formats. DocBook is a popular UNIX documentation markup format which can be translated to HTML, PDF and other presentation formats.

AsciiDoc is a plain text human readable/writable document format that can be translated to DocBook or HTML using the `asciidoc(1)` command. You can then either use `asciidoc(1)` generated HTML directly or run `asciidoc(1)` DocBook output through your favorite DocBook toolchain or use the *AsciiDoc* `a2x(1)` toolchain wrapper to produce PDF, man page, HTML and other presentation formats.

The *AsciiDoc* format is a useful presentation format in its own right: *AsciiDoc* files are unencumbered by markup and are easily viewed, proofed and edited.

AsciiDoc is light weight: it consists of a single Python script and a bunch of configuration files. Apart from `asciidoc(1)` and a Python interpreter, no other programs are required to convert *AsciiDoc* text files to DocBook or HTML. See Example *AsciiDoc* Documents below.

You write an *AsciiDoc* document the same way you would write a normal text document, there are no markup tags or arcane notations. Built-in *AsciiDoc* formatting rules have been kept to a minimum and are reasonably obvious.

Text markup conventions tend to be a matter of (often strong) personal preference: if the default syntax is not to your liking you can define your own by editing the text based `asciidoc(1)` configuration files. You can create your own configuration files to translate *AsciiDoc* documents to almost any SGML/XML markup.

`asciidoc(1)` comes with a set of configuration files to translate *AsciiDoc* articles, books or man pages to HTML or DocBook backend formats.

My AsciiDoc Itch

DocBook has emerged as the defacto standard Open Source documentation format. But DocBook is a complex language, the marked up text is difficult to read and even more difficult to write directly — I found I was spending more time typing markup tags, consulting reference manuals and fixing syntax errors, than I was writing the documentation.

Getting Started

Installing AsciiDoc

See the `README` and `INSTALL` files for install prerequisites and procedures. Packagers take a look at Appendix B: Packager Notes.

Example AsciiDoc Documents

The best way to quickly get a feel for *AsciiDoc* is to view the *AsciiDoc* web site and/or distributed examples:

- Take a look at the linked examples on the *AsciiDoc* web site home page http://www.methods.co.nz/asciidoc/. Press the *Page Source* sidebar menu item to view corresponding *AsciiDoc* source.
- Read the `*.txt` source files in the distribution `./doc` directory in conjunction with the corresponding HTML and DocBook XML files.

AsciiDoc Document Types

There are three types of *AsciiDoc* documents: article, book and manpage. All document types share the same *AsciiDoc* format with some minor variations.

Use the `asciidoc(1) -d` (`–doctype`) option to specify the *AsciiDoc* document type — the default document type is *article*.

By convention the `.txt` file extension is used for *AsciiDoc* document source files.

article

Used for short documents, articles and general documentation. See the *AsciiDoc* distribution `./doc/article.txt` example.

book

Books share the same format as articles; in addition there is the option to add level 0 book part sections.

Book documents will normally be used to produce DocBook output since DocBook processors can automatically generate footnotes, table of contents, list of tables, list of figures, list of examples and indexes.

AsciiDoc markup supports standard DocBook frontmatter and backmatter special sections (dedication, preface, bibliography, glossary, index, colophon) plus footnotes and index entries.

Example book documents

Book
: The `./doc/book.txt` file in the *AsciiDoc* distribution.

Multi-part book
: The `./doc/book-multi.txt` file in the *AsciiDoc* distribution.

manpage

Used to generate UNIX manual pages. *AsciiDoc* manpage documents observe special header title and section naming conventions — see the Manpage Documents section for details.

See also the `asciidoc(1)` man page source (`./doc/asciidoc.1.txt`) from the *AsciiDoc* distribution.

AsciiDoc Backends

The `asciidoc(1)` command translates an *AsciiDoc* formatted file to the backend format specified by the `-b` (`–backend`) command-line option. `asciidoc(1)` itself has little intrinsic knowledge of backend formats, all translation rules are contained in customizable cascading configuration files.

AsciiDoc ships with the following predefined backend output formats:

docbook

AsciiDoc generates the following DocBook document types: article, book and refentry (corresponding to the *AsciiDoc article*, *book* and *manpage* document types).

DocBook documents are not designed to be viewed directly. Most Linux distributions come with conversion tools (collectively called a toolchain) for converting DocBook files to presentation formats such as Postscript, HTML, PDF, DVI, roff (the native man page format), HTMLHelp, JavaHelp and text.

- The `–backend=docbook` command-line option produces DocBook XML. You can produce the older DocBook SGML format using the `–attribute sgml` command-line option.
- Use the optional `encoding` attribute to set the character set encoding.
- Use the optional `imagesdir` attribute to prepend to the target file name paths in image inline and block macros. Defaults to a blank string.
- The *AsciiDoc Preamble* element generates a DocBook book *preface* element although it's more usual to use an explicit *Preface* special section (see the `./doc/book.txt` example book).

xhtml11

The default `asciidoc(1)` backend is `xhtml11` which generates XHTML 1.1 markup styled with CSS2. Default output file have a `.html` extension. *xhtml11* document generation is influenced by the following optional attributes (the default behavior is to generate XHTML with no section numbers, embedded CSS and no linked admonition icon images):

numbered
: Adds section numbers to section titles.

toc
: Adds a table of contents to the start of the document.

- JavaScript needs to be enabled in your browser for this to work.
- By default *AsciiDoc* automatically embeds the required `toc.js` JavaScript in the output document — use the *linkcss* attribute to link the script.
- The following example generates a numbered table of contents by embedding the `toc.js` script in the `mydoc.html` output document (to link the script to the output document use the *linkcss* and *scriptsdir* attributes):

  ```
  $ asciidoc -a toc -a numbered mydoc.txt
  ```

toclevels
: Sets the number of title levels (1..4) reported in the table of contents (see the *toc* attribute above). Defaults to 2 and must be used with the *toc* attribute. Example usage:

```
$ asciidoc -a toc -a toclevels=3 doc/asciidoc.txt
```

linkcss
: Link CSS stylesheets and JavaScripts (see the *stylesdir* and *scriptsdir* attributes below). By default *linkcss* is undefined in which case stylesheets and scripts are automatically embedded in the output document.

stylesdir
: The name of the directory containing linked stylesheets. Defaults to `.` (the same directory as the linking document).

scriptsdir
: The name of the directory containing linked JavaScripts. Defaults to `.` (the same directory as the linking document).

icons
: Link admonition paragraph and admonition block icon images and badge images. By default *icons* is undefined and text is used in place of icon images.

iconsdir
: The name of the directory containing linked admonition and navigation icons. Defaults to `./images/icons`.

imagesdir
: This attribute is prepended to the target image file name paths in image inline and block macros. Defaults to a blank string.

theme
: Use alternative stylesheets (see Stylesheets).

badges
: Link badges (*XHTML 1.1*, *CSS* and *Get Firefox!*) in document footers. By default badges are omitted

badges is undefined).

The path names of images, icons and scripts are relative to the output document not the source document.

encoding
: Set the input and output document character set encoding. For example the –`attribute encoding=ISO-8859-1` command-line option will set the character set encoding to `ISO-8859-1`.

- The default encoding is UTF-8.
- This attribute specifies the character set in the output document.
- The encoding name must correspond to a Python codec name or alias.
- The *encoding* attribute can be set using an AttributeEntry inside the document header but it must come at the start of the document before the document title. For example:

```
:encoding: ISO-8859-1
```

quirks
: Use the `xhtml11-quirks.css` stylesheet to work around IE6 browser incompatibilities (this is the default behavior).

Stylesheets

AsciiDoc XHTML output is styled using CSS2 stylesheets from the distribution `./stylesheets/` directory.

All browsers have CSS quirks, but Microsoft's IE6 has so many omissions and errors that the `xhtml11-quirks.css` stylesheet and `xhtml11-quirks.conf` configuration files are included during XHTML backend processing to to implement work-arounds for IE6. If you don't use IE6 then the quirks stylesheet and configuration files can be omitted using the –`attribute quirks!` command-line option.

Default *xhtml11* stylesheets:

`./stylesheets/xhtml11.css`
: The main stylesheet.

`./stylesheets/xhtml11-manpage.css`
: Tweaks for manpage document type generation.

`./stylesheets/xhtml11-quirks.css`
: Stylesheet modifications to work around IE6 browser incompatibilities.

Use the *theme* attribute to select and alternative set of stylesheets. For example, the command-line option `-a theme=foo` will use stylesheets `foo.css`, `foo-manpage.css` and `foo-quirks.css`.

html4

This backend generates plain (unstyled) HTML 4.01 Transitional markup.

linuxdoc

The *AsciiDoc* linuxdoc backend is still distributed but is no longer being actively developed or tested with new *AsciiDoc* releases (the last supported release was *AsciiDoc* 6.0.3).

- Tables are not supported.
- Images are not supported.
- Callouts are not supported.
- Horizontal labeled lists are not supported.
- Only article document types are allowed.
- The Abstract section can consist only of a single paragraph.
- An *AsciiDoc* Preamble is not allowed.
- The LinuxDoc output format does not support multiple labels per labeled list item although LinuxDoc conversion programs generally output all the labels with a warning.
- Don't apply character formatting to the `link` macro attributes, LinuxDoc does not allow displayed link text to be formatted.

The default output file name extension is `.sgml`.

latex

An experimental LaTeX backend has been written for *AsciiDoc* by Benjamin Klum. A tutorial `./doc/latex-backend.html` is included in the *AsciiDoc* distribution which can also be viewed at http://www.methods.co.nz/asciidoc/latex-backend.html.

Document Structure

An *AsciiDoc* document consists of a series of block elements starting with an optional document Header, followed by an optional Preamble, followed by zero or more document Sections.

Almost any combination of zero or more elements constitutes a valid *AsciiDoc* document: documents can range from a single sentence to a multi-part book.

Block Elements

Block elements consist of one or more lines of text and may contain other block elements.

The *AsciiDoc* block structure can be informally summarized [1] as follows:

```
Document      ::= (Header?,Preamble?,Section*)
Header        ::= (Title,(AuthorLine,RevisionLine?)?)
AuthorLine    ::= (FirstName,(MiddleName?,LastName)?,EmailAddress?)
RevisionLine  ::= (Revision?,Date)
Preamble      ::= (SectionBody)
Section       ::= (Title,SectionBody?,(Section)*)
SectionBody   ::= ((BlockTitle?,Block)|BlockMacro)+
Block         ::= (Paragraph|DelimitedBlock|List|Table)
List          ::= (BulletedList|NumberedList|LabeledList|CalloutList)
BulletedList  ::= (ListItem)+
NumberedList  ::= (ListItem)+
CalloutList   ::= (ListItem)+
LabeledList   ::= (ItemLabel+,ListItem)+
ListItem      ::= (ItemText,(List|ListParagraph|ListContinuation)*)
Table         ::= (Ruler,TableHeader?,TableBody,TableFooter?)
TableHeader   ::= (TableRow+,TableUnderline)
TableFooter   ::= (TableRow+,TableUnderline)
TableBody     ::= (TableRow+,TableUnderline)
TableRow      ::= (TableData+)
```

Where:

- *?* implies zero or one occurrence, + implies one or more occurrences, * implies zero or more occurrences.
- All block elements are separated by line boundaries.
- `BlockId`, `AttributeEntry` and `AttributeList` block elements (not shown) can occur almost anywhere.
- There are a number of document type and backend specific restrictions imposed on the block syntax.
- The following elements cannot contain blank lines: Header, Title, Paragraph, ItemText.
- A ListParagraph is a Paragraph with its *listelement* option set.
- A ListContinuation is a list continuation element.

Header

The Header is optional but must start on the first line of the document and must begin with a document title. Optional Author and Revision lines immediately follow the title. The header can be preceded by a CommentBlock or comment lines.

The author line contains the author's name optionally followed by the author's email address. The author's name consists of a first name followed by optional middle and last names separated by white space. Multi-word first, middle and last names can be entered in the header author line using the underscore as a word separator. The email address comes last and must be enclosed in angle <> brackets. Author names

[1] This is a rough structural guide, not a rigorous syntax definition

cannot contain angle <> bracket characters.

The optional document header revision line should immediately follow the author line. The revision line can be one of two formats:

1. A an alphanumeric document revision number followed by a date:
   - The revision number and date must be separated by a comma.
   - The revision number is optional but must contain at least one numeric character.
   - Any non-numeric characters preceding the first numeric character will be dropped.
2. An RCS/CSV/SVN $Id$ marker.

The document heading is separated from the remainder of the document by one or more blank lines.

Here's an example *AsciiDoc* document header:

```
Writing Documentation using AsciiDoc
====================================
Stuart Rackham <srackham@methods.co.nz>
v2.0, February 2003
```

You can override or set header parameters by passing *revision*, *data*, *email*, *author*, *authorinitials*, *firstname* and *lastname* attributes using the `asciidoc(1) -a` (–`attribute`) command-line option. For example:

```
$ asciidoc -a date=2004/07/27 article.txt
```

Attributes can also be added to the header for substitution in the header template with Attribute Entry elements.

Preamble

The Preamble is an optional untitled section body between the document Header and the first Section title.

Sections

AsciiDoc supports five section levels 0 to 4 (although only book documents are allowed to contain level 0 sections). Section levels are delineated by the section titles.

Sections are translated using configuration file markup templates. To determine which configuration file template to use *AsciiDoc* first searches for special section titles in the `[specialsections]` configuration entries, if not found it uses the `[sect<level>]` template.

The `-n` (–`section-numbers`) command-line option auto-numbers HTML outputs (DocBook line numbering is handled automatically by the DocBook toolchain commands).

Section IDs are auto-generated from section titles if the `sectids` attribute is defined (the default behavior). The primary purpose of this feature is to ensure persistence of table of contents links: missing section IDs are generated dynamically by the JavaScript TOC generator **after** the page is loaded. This means, for example, that if you go to a bookmarked dynamically generated TOC address the page will load but the browser will ignore the (as yet ungenerated) section ID.

The IDs are generated by the following algorithm:

- Replace all non-alphanumeric title characters with an underscore.
- Strip leading or trailing underscores.
- Convert to lowercase.
- Prepend an underscore (so there's no possibility of name clashes with existing document IDs).
- A numbered suffix (`_2`, `_3` …) is added if a same named auto-generated section ID exists.

For example the title *Jim's House* would generate the ID `_jim_s_house`.

Special Sections

In addition to normal sections, documents can contain optional frontmatter and backmatter sections — for example: preface, bibliography, table of contents, index.

The *AsciiDoc* configuration file `[specialsections]` section specifies special section titles and the corresponding backend markup templates.

`[specialsections]` entries are formatted like:

```
<pattern>=<name>
```

`<pattern>` is a Python regular expression and `<name>` is the name of a configuration file markup template section. If the `<pattern>` matches an *AsciiDoc* document section title then the backend output is marked up using the `<name>` markup template (instead of the default `sect<level>` section template). The {title} attribute value is set to the value of the matched regular expression group named *title*, if there is no *title* group {title} defaults to the the whole of the *AsciiDoc* section title.

AsciiDoc comes preconfigured with the following special section titles:

```
Preface                    (book documents only)
Abstract                   (article documents only)
Dedication                 (book documents only)
Glossary
Bibliography|References
Colophon                   (book documents only)
Index
Appendix [A-Z][:.] <title>
```

Inline Elements

Inline document elements are used to markup character formatting and various types of text substitution. Inline elements and inline element syntax is defined in the `asciidoc(1)` configuration files.

Here is a list of *AsciiDoc* inline elements in the (default) order in which they are processed:

Special characters
: These character sequences escape special characters used by the backend markup (typically "<", ">", and "&"). See `[specialcharacters]` configuration file sections.

Quotes
: Characters that markup words and phrases; usually for character formatting. See `[quotes]` configuration file sections.

Special Words
: Word or word phrase patterns singled out for markup without the need for further annotation. See `[specialwords]` configuration file sections.

Replacements
: Each Replacement defines a word or word phrase pattern to search for along with corresponding replacement text. See `[replacements]` configuration file sections.

Attributes
: Document attribute names enclosed in braces (attribute references) are replaced by the corresponding attribute value.

Inline Macros
: Inline macros are replaced by the contents of parametrized configuration file sections.

Document Processing

The *AsciiDoc* source document is read and processed as follows:

1. The document *Header* is parsed, header parameter values are substituted into the configuration file `[header]` template section which is then written to the output file.

2. Each document *Section* is processed and its constituent elements translated to the output file.

3. The configuration file `[footer]` template section is substituted and written to the output file.

When a block element is encountered `asciidoc(1)` determines the type of block by checking in the following order (first to last): (section) Titles, BlockMacros, Lists, DelimitedBlocks, Tables, AttributeEntrys, AttributeLists, BlockTitles, Paragraphs.

The default paragraph definition `[paradef-default]` is last element to be checked.

Knowing the parsing order will help you devise unambiguous macro, list and block syntax rules.

Inline substitutions within block elements are performed in the following default order:

1. Special characters

2. Quotes

3. Special words

4. Replacements
5. Attributes
6. Inline Macros
7. Passthroughs
8. Replacements2

The substitutions and substitution order performed on Title, Paragraph and DelimitedBlock elements is determined by configuration file parameters.

Text Formatting

Quoted Text

Words and phrases can be formatted by enclosing inline text with quote characters:

Emphasized text
: Word phrases 'enclosed in single quote characters' (acute accents) or _underline characters_ are emphasized.

Strong text
: Word phrases *enclosed in asterisk characters* are rendered in a strong font (usually bold).

`Monospaced text`
: Word phrases `enclosed in backtick characters` (grave accents) or +plus characters+ are rendered in a monospaced font.

“Quoted text”
: Phrases ``enclosed with two grave accents to the left and two acute accents to the right'' are rendered in quotation marks.

Unquoted text
: Placing #hashes around text# does nothing, it is a mechanism to allow inline attributes to be applied to otherwise unformatted text (see example below).

The alternative underline and plus characters, while marginally less readable, are arguably a better choice than the backtick and apostrophe characters as they are not normally used for, and so not confused with, punctuation.

Quoted text can be prefixed with an attribute list. Currently the only use made of this feature is to allow the font color, background color and size to be specified (XHTML/HTML only, not DocBook) using the first three positional attribute arguments. The first argument is the text color; the second the background color; the third is the font size. Colors are valid CSS colors and the font size is a number which treated as em units. Here are some examples:

```
[red]#Red text#.
[,yellow]*bold text on a yellow background*.
```

```
[blue,#b0e0e6]+Monospaced blue text on a light blue background+
[,,2]#Double sized text#.
```

New quotes can be defined by editing `asciidoc(1)` configuration files. See the Configuration Files section for details.

Quoted text properties

- Quoting cannot be overlapped.
- Different quoting types can be nested.
- To suppress quoted text formatting place a backslash character immediately in front of the leading quote character(s). In the case of ambiguity between escaped and non-escaped text you will need to escape both leading and trailing quotes, in the case of multi-character quotes you may even need to escape individual characters.
- A configuration file `[quotes]` entry can be subsequently undefined by setting it to a blank value.

Constrained and Unconstrained Quotes

There are actually two types of quotes:

Constrained quotes

Quote text that must be bounded by white space, for example a phrase or a word. These are the most common type of quote and are the ones discussed previously.

Unconstrained quotes

Unconstrained quotes have no boundary constraints and can be placed anywhere within inline text. For consistency and to make them easier to remember unconstrained quotes are double-ups of the _, `*`, + and `#` constrained quotes:

```
__unconstrained emphasized text__
**unconstrained strong text**
++unconstrained monospaced text++
##unconstrained unquoted text##
```

The following example emboldens the letter F:

```
**F**ile Open...
```

Inline Passthroughs

This special text quoting mechanism passes inline text to the output document without the usual substitutions. There are two flavors:

+++Triple-plus passthrough+++

No change is made to the quoted text, it is passed verbatim to the output document.

$$Double-dollar passthrough$$
: Special characters are escaped but no other changes are made. This passthrough can be prefixed with inline attributes.

Superscripts and Subscripts

Put ^carets on either^ side of the text to be superscripted, put ~tildes on either side~ of text to be subscripted. For example, the following line:

```
e^{amp}#960;i^+1 = 0. H~2~O and x^10^. Some ^super text^
and ~some sub text~
```

Is rendered like:

$e^{\#i}$+1 = 0. H_2O and x^{10}. Some $^{\text{super text}}$ and $_{\text{some sub text}}$

Superscripts and subscripts are implemented as unconstrained quotes so they can be escaped with a leading backslash and prefixed with with an attribute list.

Line Breaks (HTML/XHTML)

A plus character preceded by at least one space character at the end of a line forces a line break. It generates an HTML line break (`<br />`) tag. Line breaks are ignored when outputting to DocBook since it has no line break element.

Rulers (HTML/XHTML)

A line of four or more apostrophe characters will generate an HTML ruler (`<hr />`) tag. Ignored when generating non-HTML output formats.

Tabs

By default tab characters input files will translated to 8 spaces. Tab expansion is set with the *tabsize* entry in the configuration file `[miscellaneous]` section and can be overridden in the *include* block macro by setting a *tabsize* attribute in the macro's attribute list. For example:

```
include::addendum.txt[tabsize=2]
```

The tab size can also be set using the attribute command-line option, for example `--attribute tabsize=4`

Replacements

The following replacements are defined in the default *AsciiDoc* configuration:

```
(C) copyright, (TM) trademark, (R) registered trademark,
-- em dash, ... ellipsis.
```

Which are rendered as:

© copyright, ™ trademark, ® registered trademark, — em dash, … ellipsis.

The Configuration Files section explains how to configure your own replacements.

Special Words

Words defined in `[specialwords]` configuration file sections are automatically marked up without having to be explicitly notated.

The Configuration Files section explains how to add and replace special words.

Titles

Document and section titles can be in either of two formats:

Two line titles

A two line title consists of a title line, starting hard against the left margin, and an underline. Section underlines consist a repeated character pairs spanning the width of the preceding title (give or take up to three characters):

The default title underlines for each of the document levels are:

```
Level 0 (top level):     ======================
Level 1:                 ----------------------
Level 2:                 ~~~~~~~~~~~~~~~~~~~~~~
Level 3:                 ^^^^^^^^^^^^^^^^^^^^^^
Level 4 (bottom level):  ++++++++++++++++++++++
```

Examples:

```
Level One Section Title
-----------------------
```

```
Level 2 Subsection Title
~~~~~~~~~~~~~~~~~~~~~~~~
```

One line titles

One line titles consist of a single line delimited on either side by one or more equals characters (the number of equals characters corresponds to the section level minus one). Here are some examples (levels 2 and 3 illustrate the optional trailing equals characters syntax):

```
= Document Title (level 0) =
== Section title (level 1) ==
=== Section title (level 2) ===
==== Section title (level 3) ====
===== Section title (level 4) =====
```

Note

- One or more spaces must fall between the title and the delimiters.
- The trailing title delimiter is optional.
- The one-line title syntax can be changed by editing the configuration file `[titles]` section `sect0`…`sect4` entries.

BlockTitles

A BlockTitle element is a single line beginning with a period followed by a title. The title is applied to the next Paragraph, DelimitedBlock, List, Table or BlockMacro. For example:

```
.Notes
- Note 1.
- Note 2.
```

is rendered as:

Notes

- Note 1.
- Note 2.

BlockId Element

A *BlockId* is a single line block element containing a unique identifier enclosed in double square brackets. It is used to assign an identifier to the ensuing block element for use by referring links. For example:

```
[[chapter-titles]]
Chapter titles can be ...
```

The preceding example identifies the following paragraph so it can be linked from other location, for example with `<<chapter-titles,chapter titles>>`.

BlockId elements can be applied to Title, Paragraph, List, DelimitedBlock, Table and BlockMacro elements. The BlockId element is really just an AttributeList with a special syntax which sets the `{id}` attribute for substitution in the subsequent block's markup template.

The *BlockId* element has the same syntax and serves a similar function to the anchor inline macro.

Paragraphs

Paragraphs are terminated by a blank line, the end of file, or the start of a DelimitedBlock.

Paragraph markup is specified by configuration file `[paradef*]` sections. *AsciiDoc* ships with the following predefined paragraph types:

Default Paragraph

A Default paragraph (`[paradef-default]`) consists of one or more non-blank lines of text. The first line must start hard against the left margin (no intervening white space). The processing expectation of the default paragraph type is that of a normal paragraph of text.

The *verse* paragraph style preserves line boundaries and is useful for lyrics and poems. For example:

```
[verse]
Consul *necessitatibus* per id,
consetetur, eu pro everti postulant
homero verear ea mea, qui.
```

Renders:

Consul **necessitatibus** per id,
consetetur, eu pro everti postulant
homero verear ea mea, qui.

Literal Paragraph

A Literal paragraph (`[paradef-literal]`) consists of one or more lines of text, where the first line is indented by one or more space or tab characters. Literal paragraphs are rendered verbatim in a monospaced font usually without any distinguishing background or border. There is no text formatting or substitutions within Literal paragraphs apart from Special Characters and Callouts. For example:

```
  Consul *necessitatibus* per id,
  consetetur, eu pro everti postulant
  homero verear ea mea, qui.
```

Renders:

```
Consul *necessitatibus* per id,
consetetur, eu pro everti postulant
homero verear ea mea, qui.
```

Admonition Paragraphs

Tip, *Note*, *Important*, *Warning* and *Caution* paragraph definitions support the corresponding DocBook admonishment elements — just write a normal paragraph but place `NOTE:`, `TIP:`, `IMPORTANT:`, `WARNING:` or `CAUTION:` as the first word of the paragraph. For example:

```
NOTE: This is an example note.
```

or the alternative syntax:

```
[NOTE]
This is an example note.
```

Renders:

This is an example note.

If your admonition is more than a single paragraph use an admonition block instead.

Admonition Icons and Captions

Admonition customization with `icons`, `iconsdir`, `icon` and `caption` attributes does not apply when generating DocBook output. If you are going the DocBook route then the `a2x(1)` –`no-icons` and –`icons-dir` options can be used to set the appropriate XSL Stylesheets parameters.

By default the `asciidoc(1)` `xhtml11` and `html4` backends generate text captions instead of icon image links. To generate links to icon images define the `icons` attribute, for example using the `-a icons` command-line option.

The `iconsdir` attribute sets the location of linked icon images.

You can override the default icon image using the `icon` attribute to specify the path of the linked image. For example:

```
[icon="./images/icons/wink.png"]
NOTE: What lovely war.
```

Use the `caption` attribute to customize the admonition captions (not applicable to `docbook` backend). The following example suppresses the icon image and customizes the caption of a NOTE admonition (undefining the `icons` attribute with `icons=None` is only necessary if admonition icons have been enabled):

```
[icons=None, caption="My Special Note"]
NOTE: This is my special note.
```

This subsection also applies to Admonition Blocks.

Delimited Blocks

Delimited blocks are blocks of text enveloped by leading and trailing delimiter lines (normally a series of four or more repeated characters). The behavior of Delimited Blocks is specified by entries in configuration file `[blockdef*]` sections.

Predefined Delimited Blocks

AsciiDoc ships with a number of predefined DelimitedBlocks (see the `asciidoc.conf` configuration file in the `asciidoc(1)` program directory):

Predefined delimited block underlines:

```
CommentBlock:     //////////////////////////
```

```
PassthroughBlock: ++++++++++++++++++++++++++
ListingBlock:     --------------------------
LiteralBlock:     ..........................
SidebarBlock:     **************************
QuoteBlock:       __________________________
ExampleBlock:     ==========================
Filter blocks:    code~~~~~~~~~~~~~~~~~~~~~~
                  source~~~~~~~~~~~~~~~~~~~~
                  music~~~~~~~~~~~~~~~~~~~~~
```

The code, source and music filter blocks are detailed in the Filters section.

Table 1. Default DelimitedBlock substitutions

	Passthrough	Listing	Literal	Sidebar	Quote
Callouts	No	Yes	Yes	No	No
Attributes	Yes	No	No	Yes	Yes
Inline Macros	Yes	No	No	Yes	Yes
Quotes	No	No	No	Yes	Yes
Replacements	No	No	No	Yes	Yes
Special chars	No	Yes	Yes	Yes	Yes
Special words	No	No	No	Yes	Yes

Listing Blocks

ListingBlocks are rendered verbatim in a monospaced font, they retain line and whitespace formatting and often distinguished by a background or border. There is no text formatting or substitutions within Listing blocks apart from Special Characters and Callouts. Listing blocks are often used for code and file listings.

Here's an example:

```
--------------------------------------
#include <stdio.h>


int main() {
    printf("Hello World!\n");
    exit(0);
}
--------------------------------------
```

Which will be rendered like:

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    exit(0);
}
```

Literal Blocks

LiteralBlocks behave just like LiteralParagraphs except you don't have to indent the contents.

LiteralBlocks can be used to resolve list ambiguity. If the following list was just indented it would be processed as an ordered list (not an indented paragraph):

```
....................
1. Item 1
2. Item 2
....................
```

Renders:

```
1. Item 1
2. Item 2
```

A *verse* style can be applied to LiteralBlocks (useful for lyrics and poems). For example:

```
[verse]
......................................
Consul *necessitatibus* per id,
consetetur, eu pro everti postulant
homero verear ea mea, qui.

Qui in magna commodo, est labitur
dolorum an. Est ne *magna primis
adolescens*.
......................................
```

Renders:

Consul **necessitatibus** per id,
consetetur, eu pro everti postulant
homero verear ea mea, qui.

Qui in magna commodo, est labitur
dolorum an. Est ne **magna primis
adolescens**.

SidebarBlocks

A sidebar is a short piece of text presented outside the narrative flow of the main text. The sidebar is normally presented inside a bordered box to set it apart from the main text.

The sidebar body is treated like a normal section body.

Here's an example:

```
.An Example Sidebar
************************************************
Any AsciiDoc SectionBody element (apart from
SidebarBlocks) can be placed inside a sidebar.
************************************************
```

Which will be rendered like:

An Example Sidebar

Any *AsciiDoc* SectionBody element (apart from SidebarBlocks) can be placed inside a sidebar.

Comment Blocks

The contents of CommentBlocks are not processed; they are useful for annotations and for excluding new or outdated content that you don't want displayed. Here's and example:

```
//////////////////////////////////////////
CommentBlock contents are not processed by
asciidoc(1).
//////////////////////////////////////////
```

See also Comment Lines.

Passthrough Blocks

PassthroughBlocks are for backend specific markup, text is only subject to attribute and macro substitution. PassthroughBlock content will generally be backend specific. Here's an example:

```
++++++++++++++++++++++++++++++++++++++
<table border="1"><tr>
  <td>Cell 1</td>
  <td>Cell 2</td>
</tr></table>
++++++++++++++++++++++++++++++++++++++
```

Quote Blocks

QuoteBlocks are used for quoted passages of text. *attribution* and *citetitle* named attributes specify the author and source of the quote (they are equivalent to positional attribute list entries 1 and 2 respectively). Both attributes are optional and the block body is treated like a SectionBody. For example:

```
[Bertrand Russell, The World of Mathematics (1956)]
____________________________________________________________________
A good notation has subtlety and suggestiveness which at times makes
it almost seem like a live teacher.
____________________________________________________________________
```

Which is rendered as:

> A good notation has subtlety and suggestiveness which at times makes it almost seem like a live teacher.
>
> — Bertrand Russell *The World of Mathematics (1956)*

In this example unquoted positional attributes have been used, the following quoted positional and named attributes are equivalent (if the attribute list contained commas then quoting would have been mandatory):

```
["Bertrand Russell","The World of Mathematics (1956)"]
[attribution="Bertrand Russell",citetitle="The World of Mathematics (1956)"]
```

You can render poems and lyrics with a combination of Quote and Literal blocks. For example:

```
[William Blake,from Auguries of Innocence]
_____________________________________________________________________
[verse]
.....................................................................
To see a world in a grain of sand,
And a heaven in a wild flower,
Hold infinity in the palm of your hand,
And eternity in an hour.
.....................................................................
_____________________________________________________________________
```

Which is rendered as:

> To see a world in a grain of sand,
> And a heaven in a wild flower,
> Hold infinity in the palm of your hand,
> And eternity in an hour.
>
> — William Blake *from Auguries of Innocence*

Example Blocks

ExampleBlocks encapsulate the DocBook Example element and are used for, well, examples. Example blocks can be titled by preceding them with a *BlockTitle*. DocBook toolchains normally number examples and generate a *List of Examples* backmatter section.

Example blocks are delimited by lines of equals characters and you can put any block elements apart from Titles, BlockTitles and Sidebars) inside an example block. For example:

```
.An example
=====================================================================
Qui in magna commodo, est labitur dolorum an. Est ne magna primis
adolescens.
=====================================================================
```

Renders:

Example 1. An example

Qui in magna commodo, est labitur dolorum an. Est ne magna primis adolescens.

The title prefix that is automatically inserted by `asciidoc(1)` can be customized with the `caption` attribute (`xhtml11` and `html4` backends). For example

```
[caption="Example 1: "]
.An example with a custom caption
=====================================================================
Qui in magna commodo, est labitur dolorum an. Est ne magna primis
adolescens.
=====================================================================
```

Admonition Blocks

The ExampleBlock definition includes a set of admonition styles (NOTE, TIP, IMPORTANT, WARNING, CAUTION) for generating admonition blocks (admonitions containing more than just a simple paragraph). Just precede the ExampleBlock with an attribute list containing the admonition style name. For example:

```
[NOTE]
.A NOTE block
======================================================================
Qui in magna commodo, est labitur dolorum an. Est ne magna primis
adolescens.

. Fusce euismod commodo velit.
. Vivamus fringilla mi eu lacus.
  .. Fusce euismod commodo velit.
  .. Vivamus fringilla mi eu lacus.
. Donec eget arcu bibendum
  nunc consequat lobortis.
======================================================================
```

Renders:

A NOTE block

Qui in magna commodo, est labitur dolorum an. Est ne magna primis adolescens.

1. Fusce euismod commodo velit.
2. Vivamus fringilla mi eu lacus.
   a. Fusce euismod commodo velit.
   b. Vivamus fringilla mi eu lacus.
3. Donec eget arcu bibendum nunc consequat lobortis.

See also Admonition Icons and Captions.

Lists

List types

- Bulleted lists. Also known as itemized or unordered lists.
- Numbered lists. Also called ordered lists.
- Labeled lists. Sometimes called variable or definition lists.
- Callout lists (a list of callout annotations).

List behavior

- Indentation is optional and does not determine nesting, indentation does however make the source more readable.
- A nested list must use a different syntax from its parent so that `asciidoc(1)` can distinguish the start of a nested list.
- By default lists of the same type can only be nested two deep; this could be increased by defining new list definitions.
- In addition to nested lists a list item will include immediately following Literal paragraphs.
- Use List Item Continuation to include other block elements in a list item.
- The `listindex` intrinsic attribute is the current list item index (1..). If this attribute is not inside a list then it's value is the number of items in the most recently closed list. Useful for displaying the number of items in a list.

Bulleted and Numbered Lists

Bulleted list items start with a dash or an asterisk followed by a space or tab character. Bulleted list syntaxes are:

```
- List item.
* List item.
```

Numbered list items start with an optional number or letter followed by a period followed by a space or tab character. List numbering is optional. Numbered list syntaxes are:

```
.  Integer numbered list item.
1. Integer numbered list item with optional numbering.
.. Lowercase letter numbered list item.
a. Lowercase letter numbered list item with optional numbering.
```

Here are some examples:

```
- Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
  * Fusce euismod commodo velit.
  * Qui in magna commodo, est labitur dolorum an. Est ne magna primis
    adolescens. Sit munere ponderum dignissim et. Minim luptatum et
    vel.
  * Vivamus fringilla mi eu lacus.
  * Donec eget arcu bibendum nunc consequat lobortis.
- Nulla porttitor vulputate libero.
  . Fusce euismod commodo velit.
  . Vivamus fringilla mi eu lacus.
    .. Fusce euismod commodo velit.
    .. Vivamus fringilla mi eu lacus.
  . Donec eget arcu bibendum nunc consequat lobortis.
- Praesent eget purus quis magna eleifend eleifend.
  1. Fusce euismod commodo velit.
    a. Fusce euismod commodo velit.
    b. Vivamus fringilla mi eu lacus.
    c. Donec eget arcu bibendum nunc consequat lobortis.
  2. Vivamus fringilla mi eu lacus.
  3. Donec eget arcu bibendum nunc consequat lobortis.
  4. Nam fermentum mattis ante.
```

Which render as:

- Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
    - Fusce euismod commodo velit.
    - Qui in magna commodo, est labitur dolorum an. Est ne magna primis adolescens. Sit munere ponderum dignissim et. Minim luptatum et vel.
    - Vivamus fringilla mi eu lacus.
    - Donec eget arcu bibendum nunc consequat lobortis.
- Nulla porttitor vulputate libero.
    1. Fusce euismod commodo velit.
    2. Vivamus fringilla mi eu lacus.
        a. Fusce euismod commodo velit.
        b. Vivamus fringilla mi eu lacus.
    3. Donec eget arcu bibendum nunc consequat lobortis.
- Praesent eget purus quis magna eleifend eleifend.
    1. Fusce euismod commodo velit.
        a. Fusce euismod commodo velit.
        b. Vivamus fringilla mi eu lacus.
        c. Donec eget arcu bibendum nunc consequat lobortis.
    2. Vivamus fringilla mi eu lacus.
    3. Donec eget arcu bibendum nunc consequat lobortis.
    4. Nam fermentum mattis ante.

Vertical Labeled Lists

Labeled list items consist of one or more text labels followed the text of the list item.

An item label begins a line with an alphanumeric character hard against the left margin and ends with a double colon `::` or semi-colon `;;`.

The list item text consists of one or more lines of text starting on the line immediately following the label and can be followed by nested List or ListParagraph elements. Item text can be optionally indented.

Here are some examples:

```
Lorem::
  Fusce euismod commodo velit.

  Fusce euismod commodo velit.

Ipsum::
  Vivamus fringilla mi eu lacus.
  * Vivamus fringilla mi eu lacus.
  * Donec eget arcu bibendum nunc consequat lobortis.
Dolor::
  Donec eget arcu bibendum nunc consequat lobortis.
  Suspendisse;;
    A massa id sem aliquam auctor.
  Morbi;;
    Pretium nulla vel lorem.
  In;;
    Dictum mauris in urna.
```

Which render as:

Lorem
: Fusce euismod commodo velit.

```
Fusce euismod commodo velit.
```

Ipsum
: Vivamus fringilla mi eu lacus.

- Vivamus fringilla mi eu lacus.
- Donec eget arcu bibendum nunc consequat lobortis.

Dolor
: Donec eget arcu bibendum nunc consequat lobortis.

Suspendisse
A massa id sem aliquam auctor.

Morbi
Pretium nulla vel lorem.

In
Dictum mauris in urna.

Horizontal Labeled Lists

Horizontal labeled lists differ from vertical labeled lists in that the label and the list item sit side-by-side as opposed to the item under the label. Item text must begin on the same line as the label although you can begin item text on the next line if you follow the label with a backslash.

The following horizontal list example also illustrates the omission of optional indentation:

```
*Lorem*:: Fusce euismod commodo velit.  Qui in magna commodo, est
labitur dolorum an. Est ne magna primis adolescens.

  Fusce euismod commodo velit.

*Ipsum*:: Vivamus fringilla mi eu lacus.
- Vivamus fringilla mi eu lacus.
- Donec eget arcu bibendum nunc consequat lobortis.

*Dolor*:: \
  - Vivamus fringilla mi eu lacus.
  - Donec eget arcu bibendum nunc consequat lobortis.
```

Which render as:

Lorem

Fusce euismod commodo velit. Qui in magna commodo, est labitur dolorum an. Est ne magna primis adolescens.

```
Fusce euismod commodo velit.
```

Ipsum

Vivamus fringilla mi eu lacus.

- Vivamus fringilla mi eu lacus.
- Donec eget arcu bibendum nunc consequat lobortis.

Dolor

- Vivamus fringilla mi eu lacus.
- Donec eget arcu bibendum nunc consequat lobortis.

- Use vertical labeled lists in preference to horizontal labeled lists — current PDF toolchains do not make a good job of determining the relative column widths.
- If you are generating DocBook markup the horizontal labeled lists should not be nested because the *DocBook XML V4.2* DTD does not permit nested informal tables (although DocBook XSL Stylesheets process them correctly).

Question and Answer Lists

AsciiDoc comes pre-configured with a labeled list for generating DocBook question and answer (Q&A) lists (`??` label delimiter). Example:

```
Question one??
        Answer one.
Question two??
        Answer two.
```

Renders:

1. Question one

   Answer one.

2. Question two

   Answer two.

Glossary Lists

AsciiDoc comes pre-configured with a labeled list (`:-` label delimiter) for generating DocBook glossary lists. Example:

```
A glossary term:-
    The corresponding definition.
A second glossary term:-
    The corresponding definition.
```

For working examples see the `article.txt` and `book.txt` documents in the *AsciiDoc* `./doc` distribution directory.

note To generate valid DocBook output glossary lists must be located in a glossary section.

Bibliography Lists

AsciiDoc comes with a predefined itemized list (+ item bullet) for generating bibliography entries. Example:

```
+ [[[taoup]]] Eric Steven Raymond. 'The Art of UNIX
  Programming'. Addison-Wesley. ISBN 0-13-142901-9.
+ [[[walsh-muellner]]] Norman Walsh & Leonard Muellner.
  'DocBook - The Definitive Guide'. O'Reilly & Associates.
  1999. ISBN 1-56592-580-7.
```

The `[[[<reference>]]]` syntax is a bibliography entry anchor, it generates an anchor named `<reference>` and additionally displays `[<reference>]` at the anchor position. For example `[[[taoup]]]` generates an anchor named `taoup` that displays `[taoup]` at the anchor position. Cite the reference from elsewhere your document using `<<taoup>>`, this displays a hyperlink (`[taoup]`) to the corresponding bibliography entry anchor.

For working examples see the `article.txt` and `book.txt` documents in the *AsciiDoc* `./doc` distribution directory.

To generate valid DocBook output bibliography lists must be located in a bibliography section.

List Item Continuation

To include subsequent block elements in list items (in addition to implicitly included nested lists and Literal paragraphs) place a separator line containing a single plus character between the list item and the ensuing list continuation element. Multiple block elements (excluding section Titles and BlockTitles) may be included in a list item using this technique. For example:

Here's an example of list item continuation:

```
1. List item one.
+
List item one continued with a second paragraph followed by an
Indented block.
+
.................
$ ls *.sh
$ mv *.sh ~/tmp
.................
+
List item one continued with a third paragraph.

2. List item two.

   List item two literal paragraph (no continuation required).

-  Nested list (item one).

   Nested list literal paragraph (no continuation required).
+
Nested list appended list item one paragraph

-  Nested list item two.
```

Renders:

1. List item one.

   List item one continued with a second paragraph followed by a Listing block.

   ```
   $ ls *.sh
   $ mv *.sh ~/tmp
   ```

   List item one continued with a third paragraph.

2. List item two.

   ```
   List item two literal paragraph (no continuation required).
   ```

   - Nested list (item one).

     ```
     Nested list literal paragraph (no continuation required).
     ```

     Nested list appended list item one paragraph

- Nested list item two.

List Block

A List block is a special delimited block containing a list element.

- All elements between in the List Block are part of the preceding list item. In this respect the List block behaves like List Item Continuation except that list items contained within the block do not require explicit + list item continuation lines:
- The block delimiter is a single line containing two dashes.
- Any block title or attributes are passed to the first element inside the block.

The List Block is useful for:

1. Lists with long multi-element list items.
2. Nesting a list within a parent list item (by default nested lists follow the preceding list item).

Here's an example of a nested list block:

```
.Nested List Block
1. List item one.
+
This paragraph is part of the preceding list item
+
--
a. This list is nested and does not require explicit item continuation.

This paragraph is part of the preceding list item

b. List item b.

This paragraph belongs to list item b.
--
+
This paragraph belongs to item 1.

2. Item 2 of the outer list.
```

Renders:

Nested List Block

1. List item one.

   This paragraph is part of the preceding list item

   a. This list is nested and does not require explicit item continuation.

    This paragraph is part of the preceding list item

  b. List item b.

    This paragraph belongs to list item b.

  This paragraph belongs to item 1.

2. Item 2 of the outer list.

Footnotes

The shipped *AsciiDoc* configuration includes the `footnote:[<text>]` inline macro for generating footnotes. The footnote text can span multiple lines. Example footnote:

```
A footnote footnote:[An example footnote.]
```

Which renders:

A footnote [2]

Footnotes are primarily useful when generating DocBook output — DocBook conversion programs render footnote outside the primary text flow.

Indexes

The shipped *AsciiDoc* configuration includes the inline macros for generating document index entries.

`indexterm:[<primary>,<secondary>,<tertiary>]`, `(((<primary>,<secondary>,<tertiary>)))`
: This inline macro generates an index term (the <secondary> and <tertiary> attributes are optional). For example `indexterm:[Tigers,Big cats]` (or, using the alternative syntax `(((Tigers,Big cats)))`. Index terms that have secondary and tertiary entries also generate separate index terms for the secondary and tertiary entries. The index terms appear in the index, not the primary text flow.

`indexterm2:[<primary>]`, `((<primary>))`
: This inline macro generates an index term that appears in both the index and the primary text flow. The `<primary>` should not be padded to the left or right with white space characters.

For working examples see the `article.txt` and `book.txt` documents in the *AsciiDoc* `./doc` distribution directory.

> Index entries only really make sense if you are generating DocBook markup — DocBook conversion programs automatically generate an index at the point an *Index* section appears in

[2] An example footnote.

source document.

Callouts

Callouts are a mechanism for annotating verbatim text (source code, computer output and user input for example). Callout markers are placed inside the annotated text while the actual annotations are presented in a callout list after the annotated text. Here's an example:

```
.MS-DOS directory listing
.....................................................
10/17/97   9:04         <DIR>    bin
10/16/97  14:11         <DIR>    DOS            <1>
10/16/97  14:40         <DIR>    Program Files
10/16/97  14:46         <DIR>    TEMP
10/17/97   9:04         <DIR>    tmp
10/16/97  14:37         <DIR>    WINNT
10/16/97  14:25             119  AUTOEXEC.BAT   <2>
 2/13/94   6:21          54,619  COMMAND.COM    <2>
10/16/97  14:25             115  CONFIG.SYS     <2>
11/16/97  17:17      61,865,984  pagefile.sys
 2/13/94   6:21           9,349  WINA20.386     <3>
.....................................................

<1> This directory holds MS-DOS.
<2> System startup code for DOS.
<3> Some sort of Windows 3.1 hack.
```

Which renders:

Example 2. MS-DOS directory listing

```
10/17/97   9:04         <DIR>    bin
10/16/97  14:11         <DIR>    DOS            ❶
10/16/97  14:40         <DIR>    Program Files
10/16/97  14:46         <DIR>    TEMP
10/17/97   9:04         <DIR>    tmp
10/16/97  14:37         <DIR>    WINNT
10/16/97  14:25             119  AUTOEXEC.BAT   ❷
 2/13/94   6:21          54,619  COMMAND.COM    ❸
10/16/97  14:25             115  CONFIG.SYS     ❹
11/16/97  17:17      61,865,984  pagefile.sys
 2/13/94   6:21           9,349  WINA20.386     ❺
```

❶ This directory holds MS-DOS.
❷❸ System startup code for DOS.
❺ Some sort of Windows 3.1 hack.

Explanation

- The callout marks are whole numbers enclosed in angle brackets that refer to an item index in the following callout list.

- By default callout marks are confined to LiteralParagraphs, LiteralBlocks and ListingBlocks (although this is a configuration file option and can be changed).
- Callout list item numbering is fairly relaxed — list items can start with `<n>`, `n>` or `>` where `n` is the optional list item number (in the latter case list items starting with a single `>` character are implicitly numbered starting at one).
- Callout lists should not be nested — start list items hard against the left margin.
- If you want to present a number inside angle brackets you'll need to escape it with a backslash to prevent it being interpreted as a callout mark.

To include callout icons in PDF files generated by `a2x(1)` you need to use the `–icons` command-line option.

Implementation Notes

Callout marks are generated by the *callout* inline macro while callout lists are generated using the *callout* list definition. The *callout* macro and *callout* list are special in that they work together. The *callout* inline macro is not enabled by the normal *macros* substitutions option, instead it has its own *callouts* substitution option.

The following attributes are available during inline callout macro substitution:

`{index}`
: The callout list item index inside the angle brackets.

`{coid}`
: An identifier formatted like `CO<listnumber>-<index>` that uniquely identifies the callout mark. For example `CO2-4` identifies the fourth callout mark in the second set of callout marks.

The `{coids}` attribute can be used during callout list item substitution — it is a space delimited list of callout IDs that refer to the explanatory list item.

Including callouts in included code

You can annotate working code examples with callouts — just remember to put the callouts inside source code comments. This example displays the `test.py` source file (containing a single callout) using the Source Code Highlighter Filter:

Example 3. AsciiDoc source

```
[python]
source~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
include::test.py[]
source~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

<1> Print statement.
```

Example 4. Included `test.py` source

```
print 'Hello World!'   # <1>
```

Macros

Macros are a mechanism for substituting parametrized text into output documents.

Macros have a *name*, a single *target* argument and an *attribute list*. The default syntax is `<name>:<target>[<attributelist>]` (for inline macros) and `<name>::<target>[<attributelist>]` (for block macros). Here are some examples:

```
http://www.methods.co.nz/asciidoc/index.html[Asciidoc home page]
include::chapt1.txt[tabsize=2]
mailto:srackham@methods.co.nz[]
```

Macro behavior

- `<name>` is the macro name. It can only contain letters, digits or dash characters and cannot start with a dash.
- The optional `<target>` cannot contain white space characters.
- `<attributelist>` is a list of attributes enclosed in square brackets.
- The attribute list is mandatory even if it contains no attributes.
- Expansion of non-system macro references can be escaped by prefixing a backslash character.
- Block macro attribute reference substitution is performed prior to macro expansion.
- The substitutions performed prior to Inline macro macro expansion are determined by the inline context.
- Macros are processed in the order they appear in the configuration file(s).
- Calls to inline macros can be nested inside different inline macros (an inline macro call cannot contain a nested call to itself).

Inline Macros

Inline Macros occur in an inline element context. Predefined Inline macros include *URLs*, *image* and *link* macros.

URLs

Standard http, https, ftp, file, mailto and callto URLs are rendered using predefined inline macros.

The default *AsciiDoc* inline macro syntax is very similar to a URL: all you need to do is append an attribute list containing an optional caption immediately following the URL. If no caption text is provided the URL itself is displayed.

Here are some examples:

```
http://www.methods.co.nz/asciidoc/[The AsciiDoc home page]
mailto:joe.bloggs@foobar.com[email Joe Bloggs]
mailto:joe.bloggs@foobar.com[]
callto:joe.bloggs[]
```

Which are rendered:

The *AsciiDoc* home page [http://www.methods.co.nz/asciidoc/]

email Joe Bloggs [mailto:joe.bloggs@foobar.com]

joe.bloggs@foobar.com [mailto:joe.bloggs@foobar.com]

joe.bloggs [callto:joe.bloggs]

If the `<target>` necessitates space characters they should be replaced by `%20`. For example `large%20image.png`.

Internal Cross References

Two *AsciiDoc* inline macros are provided for creating hypertext links within an *AsciiDoc* document. You can use either the standard macro syntax or the (preferred) alternative.

anchor

Used to specify hypertext link targets:

```
[[<id>,<xreflabel>]]
anchor:<id>[<xreflabel>]
```

The `<id>` is a unique identifier that must begin with a letter. The optional `<xreflabel>` is the text to be displayed by captionless *xref* macros that refer to this anchor. The optional `<xreflabel>` is only really useful when generating DocBook output. Example anchor:

```
[[X1]]
```

You may have noticed that the syntax of this inline element is the same as that of the BlockId block element, this is no coincidence since they are functionally equivalent.

xref

Creates a hypertext link to a document anchor.

```
<<<id>,<caption>>>
```

```
xref:<id>[<caption>]
```

The `<id>` refers to an existing anchor `<id>`. The optional `<caption>` is the link's displayed text. Example:

```
<<X21,attribute lists>>
```

If `<caption>` is not specified then the displayed text is auto-generated:

- The *AsciiDoc* `xhtml11` backend displays the `<id>` enclosed in square brackets.
- If DocBook is produced the DocBook toolchain is responsible for the displayed text which will normally be the referenced figure, table or section title number followed by the element's title text.

Here is an example:

```
[[tiger_image]]
.Tyger tyger
image::tiger.png[]

This can be seen in <<tiger_image>>.
```

Linking to Local Documents

Hypertext links to files on the local filesystem are specified using the *link* inline macro.

```
link:<target>[<caption>]
```

The *link* macro generates relative URLs. The link macro `<target>` is the target file name (relative to the file system location of the referring document). The optional `<caption>` is the link's displayed text. If `<caption>` is not specified then `<target>` is displayed. Example:

```
link:downloads/foo.zip[download foo.zip]
```

You can use the `<filename>#<id>` syntax to refer to an anchor within a target document but this usually only makes sense when targeting HTML documents.

Images can serve as hyperlinks using the `image` macro.

Images

Inline images are inserted into the output document using the *image* macro. The inline syntax is:

```
image:<target>[<attributes>]
```

The contents of the image file `<target>` is displayed. To display the image its file format must be supported by the target backend application. HTML and DocBook applications normally support PNG or JPG files.

`<target>` file name paths are relative to the location of the referring document.

Image macro attributes

- The optional first positional attribute list entry specifies the alternative text which is displayed if the output application is unable to process the image file. For example:

  ```
  image:images/logo.png[Company Logo]
  ```

- The optional `width` and `height` named attributes scale the image size and can be used in any combination. The following example scales the previous example to a height of 32 pixels:

  ```
  image:images/logo.png["Company Logo",height=32]
  ```

- The optional `link` named attribute is used to link the image to an external document. The following example links a screenshot thumbnail to a full size version:

  ```
  image:screen-thumbnail.png[height=32,link="screen.png"]
  ```

Block Macros

A Block macro reference must be contained in a single line separated either side by a blank line or a block delimiter.

Block macros behave just like Inline macros, with the following differences:

- They occur in a block context.
- The default syntax is `<name>::<target>[<attributelist>]` (two colons, not one).
- Markup template section names end in `-blockmacro` instead of `-inlinemacro`.

Block Identifier

The Block Identifier macro sets the `id` attribute and has the same syntax as the anchor inline macro since it performs essentially the same function — block templates employ the `id` attribute as a block link target. For example:

```
[[X30]]
```

This is equivalent to the `[id="X30"]` block attribute list.

Images

Formal titled images are inserted into the output document using the *image* macro. The syntax is:

```
image::<target>[<attributes>]
```

The block `image` macro has the same macro attributes as its inline counterpart.

Images can be titled by preceding the `image` macro with a *BlockTitle*. DocBook toolchains normally number examples and generate a *List of Figures* backmatter section.

For example:

```
.Main circuit board
image::images/layout.png[J14P main circuit board]
```

`xhtml11` and `html4` backends precede the title with a `Figure :` prefix. You can customize this prefix with the `caption` attribute. For example:

```
.Main circuit board
[caption="Figure 2:"]
image::images/layout.png[J14P main circuit board]
```

Comment Lines

Single lines starting with two forward slashes hard up against the left margin are treated as comments and are stripped from the output. Comment lines have been implemented as a block macro and are only valid in a block context — they are not treated as comments inside paragraphs or delimited blocks. Example comment line:

```
// This is a comment.
```

See also Comment Blocks.

System Macros

System macros are block macros that perform a predefined task which is hardwired into the `asciidoc(1)` program.

- You can't escape system macros with a leading backslash character (as you can with other macros).
- The syntax and tasks performed by system macros is built into `asciidoc(1)` so they don't appear in configuration files. You can however customize the syntax by adding entries to a configuration file `[macros]` section.

Include Macros

The `include` and `include1` system macros to include the contents of a named file into the source document.

The `include` macro includes a file as if it were part of the parent document — tabs are expanded and system macros processed. The contents of `include1` files are not subject to tab expansion or system macro processing nor are attribute or lower priority substitutions performed. The `include1` macro's main use is to include verbatim embedded CSS or scripts into configuration file headers. Example:

```
include::chapter1.txt[tabsize=4]
```

Include macro behavior

- If the included file name is specified with a relative path then the path is relative to the location of the referring document.
- Include macros can appear inside configuration files.
- Files included from within `DelimitedBlocks` are read to completion to avoid false end-of-block underline termination.
- File inclusion is limited to a depth of 5 to catch recursive loops.
- Attribute references are expanded inside the include `target`; if an an attribute is undefined then the included file is silently skipped.
- The *tabsize* macro attribute sets the the number of space characters to be used for tab expansion in the included file.

Conditional Inclusion Macros

Lines of text in the source document can be selectively included or excluded from processing based on the the existence (or not) of a document attribute. There are two forms of conditional inclusion macro usage, the first includes document text between the `ifdef` and `endif` macros if a document attribute is defined:

```
ifdef::<attribute>[]
:
endif::<attribute>[]
```

The second for includes document text between the `ifndef` and `endif` macros if the attribute is not defined:

```
ifndef::<attribute>[]
:
endif::<attribute>[]
```

`<attribute>` is an attribute name which is optional in the trailing `endif` macro.

Take a look at the `*.conf` configuration files in the *AsciiDoc* distribution for examples of conditional inclusion macro usage.

Two types of conditional inclusion

Conditional inclusion macros are evaluated when they are read, but there is another type of conditional inclusion based on attribute references, the latter being evaluated when the output file is written.

These examples illustrate the two forms of conditional inclusion. The only difference between them is that the first is evaluated at program load time while the second is evaluated when the output is written:

```
ifdef::world[]
```

```
  Hello World!
endif::world[]
```

```
{world#}Hello World!
```

In this example when the `{world#}` conditional attribute reference is evaluates to a zero length string if `world` is defined; if `world` is not defined the whole line is dropped.

The subtle difference between the two types of conditional inclusion has implications for *AsciiDoc* configuration files: *AsciiDoc* has to read the configuration files **before** reading the source document, this is necessary because the *AsciiDoc* source syntax is mostly defined by the configuration files. This means that any lines of markup enveloped by conditional inclusion macros will be included or excluded **before** the attribute entries in the *AsciiDoc* document header are read, so setting related attributes in the *AsciiDoc* source document header will have no effect. If you need to control configuration file markup inclusion with attribute entries in the *AsciiDoc* source file header you need to use attribute references to control inclusion instead of conditional inclusion macros (attribute references are substituted at the time the output is written rather than at program startup).

eval, sys and sys2 System Macros

These block macros exhibit the same behavior as their same named system attribute references. The difference is that system macros occur in a block macro context whereas system attributes are confined to an inline context where attribute substitution is enabled.

The following example displays a long directory listing inside a literal block:

```
------------------
sys::[ls -l *.txt]
------------------
```

Template System Macro

The `template` block macro allows the inclusion of one configuration file template section within another. The following example includes the `[admonitionblock]` section in the `[admonitionparagraph]` section:

```
[admonitionparagraph]
template::[admonitionblock]
```

Template macro behavior

- The `template::[]` macro is useful for factoring configuration file markup.
- `template::[]` macros cannot be nested.
- `template::[]` macro expansion is applied to all sections after all configuration files have been read.

Macro Definitions

Each entry in the configuration `[macros]` section is a macro definition which can take one of the following forms:

`<pattern>=<name>`
: Inline macro definition.

`<pattern>=#<name>`
: Block macro definition.

`<pattern>=+<name>`
: System macro definition.

`<pattern>`
: Delete the existing macro with this `<pattern>`.

`<pattern>` is a Python regular expression and `<name>` is the name of a markup template. If `<name>` is omitted then it is the value of the regular expression match group named *name*.

Here's what happens during macro substitution

- Each contextually relevant macro *pattern* from the `[macros]` section is matched against the input source line.
- If a match is found the text to be substituted is loaded from a configuration markup template section named like `<name>-inlinemacro` or `<name>-blockmacro` (depending on the macro type).
- Global and macro attribute list attributes are substituted in the macro's markup template.
- The substituted template replaces the macro reference in the output document.

Tables

Tables are the most complex *AsciiDoc* elements and this section is quite long. [3]

note

AsciiDoc generates nice HTML tables, but the current crop of DocBook toolchains render tables with varying degrees of success. Use tables only when really necessary.

Example Tables

The following annotated examples are all you'll need to start creating your own tables.

[3] The current table syntax is overly complicated and unwieldy to edit, hopefully a more usable syntax will appear in future versions of *AsciiDoc*.

The only non-obvious thing you'll need to remember are the column stop characters:

- Backtick (`) — align left.
- Single quote (') — align right.
- Period (.) — align center.

Simple table:

```
`---`---
1   2
3   4
5   6
--------
```

Output:

1	2
3	4
5	6

Table with title, header and footer:

```
.An example table
[grid="all"]
'---------.--------------
Column 1   Column 2
-------------------------
1          Item 1
2          Item 2
3          Item 3
-------------------------
6          Three items
-------------------------
```

Output:

Table 2. An example table

Column 1	Column 2
1	Item 1
2	Item 2
3	Item 3
6	**Three items**

Four columns totaling 15% of the *pagewidth*, CSV data:

~~~
[frame="all"]
````~15
~~~

```
1,2,3,4
a,b,c,d
A,B,C,D
~~~~~~~~
```

Output:

1	2	3	4
a	b	c	d
A	B	C	D

A table with a numeric ruler and externally sourced CSV data:

```
 [frame="all", grid="all"]
 .15`20`25`20`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ID,Customer Name,Contact Name,Customer Address,Phone
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 include::customers.csv[]
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Renders:

ID	**Customer Name**	**Contact Name**	**Customer Address**	**Phone**
AROUT	Around the Horn	Thomas Hardy	120 Hanover Sq. London	(171) 555-7788
BERGS	Berglunds snabbkop	Christina Berglund	Berguvsvagen 8 Lulea	0921-12 34 65
BLAUS	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57 Mannheim	0621-08460
BLONP	Blondel pere et fils	Frederique Citeaux	24, place Kleber Strasbourg	88.60.15.31
BOLID	Bolido Comidas preparadas	Martin Sommer	C/ Araquil, 67 Madrid	(91) 555 22 82
BONAP	Bon app'	Laurence Lebihan	12, rue des Bouchers Marseille	91.24.45.40
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	23 Tsawassen Blvd. Tsawassen	(604) 555-4729
BSBEV	B's Beverages	Victoria Ashworth	Fauntleroy Circus London	(171) 555-1212
CACTU	Cactus Comidas para llevar	Patricio Simpson	Cerrito 333 Buenos Aires	(1) 135-5555

AsciiDoc Table Block Elements

This sub-section details the *AsciiDoc* table format.

```
Table  ::= (Ruler,Header?,Body,Footer?)
Header ::= (Row+,Underline)
Footer ::= (Row+,Underline)
Body   ::= (Row+,Underline)
Row    ::= (Data+)
```

A table is terminated when the table underline is followed by a blank line or an end of file. Table underlines which separate table headers, bodies and footers should not be followed by a blank line.

Ruler

The first line of the table is called the *Ruler*. The Ruler specifies which configuration file table definition to use, column widths, column alignments and the overall table width.

There are two ruler formats:

Character ruler
: The column widths are determined by the number of table fill characters between column stop characters.

Numeric ruler
: The column widths are specified numerically. If a column width is omitted the previous width is used. In the degenerate case of no widths being specified columns are allocated equal widths.

The ruler format can be summarized as:

```
ruler ::= ((colstop,colwidth?,fillchar*)+, fillchar+, tablewidth?
```

- The *ruler* starts with a column stop character (designating the start of the first column).
- Column stop characters specify the start and alignment of each column:
  - Backtick (`) — align left.
  - Single quote (') — align right.
  - Period (.) — align center.
- In the case of *fixed* format tables the ruler column widths specify source row data column boundaries.
- The optional *tablewidth* is a number representing the size of the output table relative to the *pagewidth*. If *tablewidth* is less than one then it is interpreted as a fraction of the page width; if it is greater than one then it is interpreted as a percentage of the page width. If *tablewidth* is not specified then the table occupies the full *pagewidth* (numeric rulers) or the relative width of the ruler compared to the *textwidth* (character rulers).

Row and Data Elements

Each table row consists of a line of text containing the same number of *Data* items as there are columns in the table,

Lines ending in a backslash character are continued on the next line.

Each *Data* item is an *AsciiDoc* substitutable string. The substitutions performed are specified by the *subs* table definition entry. Data cannot contain *AsciiDoc* block elements.

The format of the row is determined by the table definition *format* value:

fixed
: Row data items are assigned by chopping the row up at ruler column width boundaries.

csv
: Data items are assigned the parsed CSV (Comma Separated Values) data.

dsv
: The DSV (Delimiter Separated Values) format is a common UNIX tabular text file format.

  - The separator character is a colon (although this can be set to any letter using the *separator* table attribute).
  - Common C-style backslash escapes are supported.
  - Blank lines are skipped.

Underline

A table *Underline* consists of a line of three or more *fillchar* characters which are end delimiters for table header, footer and body sections.

Attribute List

The following optional table attributes can be specified in an AttributeList preceding the table:

separator
: The default DSV format colon separator can be changed using the *separator* attribute. For example: `[separator="|"]`.

frame
: Defines the table border and can take the following values: *topbot* (top and bottom), *all* (all sides), *none* and *sides* (left and right sides). The default value is *topbot*.

grid
: Defines which ruler lines are drawn between table rows and columns. The *grid* attribute value can be any of the following values: *none*, *cols*, *rows* and *all*. The default value is *none*. For example `[frame="all", grid="none"]`.

format, tablewidth
: See Markup Attributes below.

You can also use an AttributeList to override the following table definition and ruler parameters: *format*, *subs*, *tablewidth*.

Markup Attributes

The following attributes are automatically available inside table tag and markup templates.

cols
: The number of columns in the table.

colalign
: Column alignment assumes one of three values (*left*, *right* or *center*). The value is determined by the corresponding ruler column stop character (only valid inside *colspec*, `headdata`, `bodydata` and `footdata` tags).

colwidth
: The output column widths are calculated integers (only valid inside *colspec*, `headdata`, `bodydata` and `footdata` tags).

colnumber
: The table column number starting at 1 (only valid inside *colspec*, `headdata`, `bodydata` and `footdata` tags).

format
: The table definition *format* value (can be overridden with attribute list entry).

tablewidth
: The ruler *tablewidth* value (can be overridden with attribute list entry).

pagewidth
: The *pagewidth* miscellaneous configuration option.

pageunits
: The *pageunits* miscellaneous configuration option.

The *colwidth* value is calculated as (`N` is the ruler column width number and `M` is the sum of the ruler column widths):

```
( N / M ) * pagewidth
```

If the ruler *tablewidth* was specified the column width is multiplied again by this value.

There is one exception: character rulers that have no *pagewidth* specified. In this case the *colwidth* value is calculated as (where `N` is the column character width measured on the table ruler):

```
( N / textwidth ) * pagewidth
```

The following attributes are available to the table markup template:

comspecs
: Expands to N substituted *comspec* tags where N is the number of columns.

headrows, footrows, bodyrows
: These references expand to sets of substituted header, footer and body rows as defined by the corresponding row and data configuration parameters.

rows
: Experimental attribute (number of source lines in table) available in table markup templates (used by experimental LaTeX backend).

Manpage Documents

Sooner or later, if you program for a UNIX environment, you're going to have to write a man page.

By observing a couple of additional conventions you can compose *AsciiDoc* files that will translate to a DocBook refentry (man page) document. The resulting DocBook file can then be translated to the native roff man page format (or other formats).

For example, the `asciidoc.1.txt` file in the *AsciiDoc* distribution `./doc` directory was used to generate both the `asciidoc.1.css-embedded.html` HTML file the `asciidoc.1` roff formatted `asciidoc(1)` man page.

Viewing and printing manpage files

Use the `man(1)` command to view the manpage file (you must include a file path even if it's only `./` otherwise `man(1)` will look for the file in the system manpage locations):

```
$ man ./asciidoc.1
```

To print a high quality man page to a postscript printer:

```
$ groff -mandoc -Tps asciidoc.1 | lpr
```

To find out more about man pages view the `man(7)` manpage (`man 7 man` command).

Document Header

A document Header is mandatory. The title line contains the man page name followed immediately by the manual section number in brackets, for example *ASCIIDOC(1)*. The title name should not contain white space and the manual section number is a single digit optionally followed by a single character.

The NAME Section

The first manpage section is mandatory, must be titled *NAME* and must contain a single paragraph (usually a single line) consisting of a list of one or more comma separated command name(s) separated from the command purpose by a dash character. The dash must have at least one white space character on either side. For example:

```
printf, fprintf, sprintf - print formatted output
```

The SYNOPSIS Section

The second manpage section is mandatory and must be titled *SYNOPSIS*.

Configuration Files

AsciiDoc source file syntax and output file markup is largely controlled by a set of cascading, text based, configuration files. At runtime The *AsciiDoc* default configuration files are combined with optional user and document specific configuration files.

Configuration File Format

Configuration files contain named sections. Each section begins with a section name in square brackets []. The section body consists of the lines of text between adjacent section headings.

- Section names consist of one or more alphanumeric, underscore or dash characters and cannot begin or end with a dash.
- Lines starting with a hash character "#" are treated as comments and ignored.
- Same named sections and section entries override previously loaded sections and section entries (this is sometimes referred to as *cascading*). Consequently, downstream configuration files need only contain those sections and section entries that need to be overridden.

When creating custom configuration files you only need to include the sections and entries that differ from the default configuration.

The best way to learn about configuration files is to read the default configuration files in the *AsciiDoc* distribution in conjunction with `asciidoc(1)` output files. You can view configuration file load sequence by turning on the `asciidoc(1) -v` (–`verbose`) command-line option.

Markup Template Sections

Markup template sections supply backend markup for translating *AsciiDoc* elements. Since the text is normally backend dependent you'll find these sections in the backend specific configuration files. A markup template section body can contain:

- Backend markup

- Attribute references
- System macro calls.
- A document content placeholder

The document content placeholder is a single | character and is replaced by text from the source element. Use the `{brvbar}` attribute reference if you need a literal | character in the template.

Special Sections

AsciiDoc reserves the following predefined special section names for specific purposes:

miscellaneous
: Configuration options that don't belong anywhere else.

attributes
: Attribute name/value entries.

specialcharacters
: Special characters reserved by the backend markup.

tags
: Backend markup tags.

quotes
: Definitions for quoted inline character formatting.

specialwords
: Lists of words and phrases singled out for special markup.

replacements, replacements2
: Find and replace substitution definitions.

specialsections
: Used to single out special section names for specific markup.

macros
: Macro syntax definitions.

titles
: Heading, section and block title definitions.

paradef*
: Paragraph element definitions.

blockdef*
: DelimitedBlock element definitions.

listdef*
: List element definitions.

tabledef*
: Table element definitions.

Each line of text in a special section is a *section entry*. Section entries share the following syntax:

name=value
: The entry value is set to *value*.

name=
: The entry value is set to a zero length string.

name
: The entry is undefined (deleted from the configuration).

Section entry behavior

- All equals characters inside the `name` must be escaped with a backslash character. If you want the `name` to end in a backslash then you need to place two backslashes at the end of the name.
- `name` and `value` are stripped of leading and trailing white space.
- Attribute names, tag entry names and markup template section names consist of one or more alphanumeric, underscore or dash characters. Names should not begin or end with a dash.
- A blank configuration file section (one without any entries) deletes any preceding section with the same name (applies to non-markup template sections).

Miscellaneous

The optional `[miscellaneous]` section specifies the following `name=value` options:

newline
: Output file line termination characters. Can include any valid Python string escape sequences. The default value is `\r\n` (carriage return, line feed). Should not be quoted or contain explicit spaces (use `\x20` instead). For example:

```
$ asciidoc -a 'newline=\n' -b docbook mydoc.txt
```

outfilesuffix
: The default extension for the output file, for example `outfilesuffix=.html`. Defaults to backend name.

tabsize
: The number of spaces to expand tab characters, for example `tabsize=4`. Defaults to 8. A *tabsize* of zero suppresses tab expansion (useful when piping included files through block filters). Included files can override this option using the *tabsize* attribute.

textwidth, pagewidth, pageunits

These global table related options are documented in the Table Configuration File Definitions sub-section.

`[miscellaneous]` configuration file entries can be set using the `asciidoc(1)` `-a` (–`attribute`) command-line option.

Titles

sectiontitle

Two line section title pattern. The entry value is a Python regular expression containing the named group *title*.

underlines

A comma separated list of document and section title underline character pairs starting with the section level 0 and ending with section level 4 underline. The default setting is:

```
underlines="==","--","~~","^^","++"
```

sect0…sect4

One line section title patterns. The entry value is a Python regular expression containing the named group *title*.

blocktitle

BlockTitle element pattern. The entry value is a Python regular expression containing the named group *title*.

subs

A comma separated list of substitutions that are performed on the document header and section titles. Defaults to *normal* substitution.

Tags

The `[tags]` section contains backend tag definitions (one per line). Tags are used to translate *AsciiDoc* elements to backend markup.

An *AsciiDoc* tag definition is formatted like `<tagname>=<starttag>|<endtag>`. For example:

```
emphasis=<em>|</em>
```

In this example `asciidoc(1)` replaces the | character with the emphasized text from the *AsciiDoc* input file and writes the result to the output file.

Use the `{brvbar}` attribute reference if you need to include a | pipe character inside tag text.

Attributes Section

The optional `[attributes]` section contains predefined attributes.

If the attribute value requires leading or trailing spaces then the text text should be enclosed in double-quote (") characters.

To delete a attribute insert a name only entry in a downstream configuration file or use the `asciidoc(1)` `–attribute name!` command-line option (the attribute name is suffixed with a ! character to delete it).

Special Characters

The `[specialcharacters]` section specifies how to escape characters reserved by the backend markup. Each translation is specified on a single line formatted like:

```
special_character=translated_characters
```

Special characters are normally confined to those that resolve markup ambiguity (in the case of SGML/XML markups the ampersand, less than and greater than characters). The following example causes all occurrences of the < character to be replaced by `&lt;`.

```
<=&lt;
```

Quoted Text

Quoting is used primarily for text formatting. The `[quotes]` section defines *AsciiDoc* quoting characters and their corresponding backend markup tags. Each section entry value is the name of a of a `[tags]` section entry. The entry name is the character (or characters) that quote the text. The following examples are taken from *AsciiDoc* configuration files:

```
[quotes]
_=emphasis
```

```
[tags]
emphasis=<em>|</em>
```

You can specify the left and right quote strings separately by separating them with a | character, for example:

```
``|''=quoted
```

Omitting the tag will disable quoting, for example, if you don't want superscripts or subscripts put the following in a custom configuration file or edit the global `asciidoc.conf` configuration file:

```
[quotes]
^=
~=
```

Unconstrained quotes are differentiated by prefixing the tag name with a hash character, for example:

```
__=#emphasis
```

Quoted text behavior

- Quote characters must be non-alphanumeric.
- To minimize quoting ambiguity try not to use the same quote characters in different quote types.

Special Words

The `[specialwords]` section is used to single out words and phrases that you want to consistently format in some way throughout your document without having to repeatedly specify the markup. The name of each entry corresponds to a markup template section and the entry value consists of a list of words and phrases to be marked up. For example:

```
[specialwords]
strongwords=NOTE: IMPORTANT:


[strongwords]
<strong>{words}</strong>
```

The examples specifies that any occurrence of `NOTE:` or `IMPORTANT:` should appear in a bold font.

Words and word phrases are treated as Python regular expressions: for example, the word `^NOTE:` would only match `NOTE:` if appeared at the start of a line.

AsciiDoc comes with three built-in Special Word types: *emphasizedwords*, *monospacedwords* and *strongwords*, each has a corresponding (backend specific) markup template section. Edit the configuration files to customize existing Special Words and to add new ones.

Special word behavior

- Word list entries must be separated by space characters.
- Word list entries with embedded spaces should be enclosed in quotation (") characters.
- A `[specialwords]` section entry of the form `name=word1 [word2…]` adds words to existing `name` entries.
- A `[specialwords]` section entry of the form `name` undefines (deletes) all existing `name` words.
- Since word list entries are processed as Python regular expressions you need to be careful to escape regular expression special characters.
- By default Special Words are substituted before Inline Macros, this may lead to undesirable consequences. For example the special word `foobar` would be expanded inside the macro call `http://www.foobar.com`. A possible solution is to emphasize whole words only by defining the word using regular expression characters, for example `\bfoobar\b`.
- If the first matched character of a special word is a backslash then the remaining characters are output without markup i.e. the backslash can be used to escape special word markup. For example the special word `\\?\b[Tt]en\b` will mark up the words `Ten` and `ten` only if they are not preceded by a backslash.

Replacements

`[replacements]` and `[replacements2]` configuration file entries specify find and replace text and are formatted like:

```
find_pattern=replacement_text
```

The find text can be a Python regular expression; the replace text can contain Python regular expression group references.

Use Replacement shortcuts for often used macro references, for example (the second replacement allows us to backslash escape the macro name):

```
NEW!=image:./images/smallnew.png[New!]
\\NEW!=NEW!
```

Replacement behavior

- The built-in replacements can be escaped with a backslash.
- If the find or replace text has leading or trailing spaces then the text should be enclosed in quotation (") characters.
- Since the find text is processed as a regular expression you need to be careful to escape regular expression special characters.
- Replacements are performed in the same order they appear in the configuration file replacements section.

Configuration File Names and Locations

Configuration files have a `.conf` file name extension; they are loaded implicitly (using predefined file names and locations) or explicitly (using the `asciidoc(1) -f` (–`conf-file`) command-line option).

Implicit configuration files are loaded from the following directories in the following order:

1. The `/etc/asciidoc` directory (if it exists).
2. The directory containing the asciidoc executable.
3. The user's `$HOME/.asciidoc` directory (if it exists).
4. The directory containing the *AsciiDoc* source file.

The following implicit configuration files from each of the above locations are loaded in the following order:

1. `asciidoc.conf`
2. `<backend>.conf`
3. `<backend>-<doctype>.conf`
4. `lang-<lang>.conf`

Where `<backend>` and `<doctype>` are values specified by the `asciidoc(1)` `-b` (–`backend`) and `-d` (–`doctype`) command-line options. `<lang>` is the value of the *AsciiDoc* `lang` attribute (defaults to `en` (English)).

Finally, configuration files named like the source file will be automatically loaded if they are found in the source file directory. For example if the source file is `mydoc.txt` and the –`backend=html4` option is used then `asciidoc(1)` will look for `mydoc.conf` and `mydoc-html4.conf` in that order.

Implicit configuration files that don't exist will be silently skipped.

The user can explicitly specify additional configuration files using the `asciidoc(1)` `-f` (–`conf-file`) command-line option. The `-f` option can be specified multiple times, in which case configuration files will be processed in the order they appear on the command-line.

For example, when we translate our *AsciiDoc* document `mydoc.txt` with:

```
$ asciidoc -f extra.conf mydoc.txt
```

Configuration files (if they exist) will be processed in the following order:

1. First default global configuration files from the asciidoc program directory are loaded:

   ```
   asciidoc.conf
   xhtml11.conf
   ```

2. Then, from the users home `~/.asciidoc` directory. This is were you put customization specific to your own asciidoc documents:

   ```
   asciidoc.conf
   xhtml11.conf
   xhtml11-article.conf
   ```

3. Next from the source document project directory (the first three apply to all documents in the directory, the last two are specific to the mydoc.txt document):

   ```
   asciidoc.conf
   xhtml11.conf
   xhtml11-article.conf
   mydoc.conf
   mydoc-xhtml11.conf
   ```

4. Finally the file specified by the `-f` command-line option is loaded:

   ```
   extra.conf
   ```

Use the `asciidoc(1) -v` (–`verbose`) command-line option to see which configuration files are loaded and the order in which they are loaded.

Document Attributes

A document attribute is comprised of a *name* and a textual *value* and is used for textual substitution in *AsciiDoc* documents and configuration files. An attribute reference (an attribute name enclosed in braces) is replaced by its corresponding attribute value.

There are four sources of document attributes (from highest to lowest precedence):

- Command-line attributes.
- AttributeEntry, AttributeList, Macro and BlockId elements.
- Configuration file `[attributes]` sections.
- Intrinsic attributes.

Within each of these divisions the last processed entry takes precedence.

If an attribute is not defined then the line containing the attribute reference is dropped. This property is used extensively in *AsciiDoc* configuration files to facilitate conditional markup generation.

Attribute Entries

The `AttributeEntry` block element allows document attributes to be assigned within an *AsciiDoc* document. Attribute entries are added to the global document attributes dictionary. The attribute name/value syntax is a single line like:

```
:<name>: <value>
```

For example:

```
:Author Initials: JB
```

This will set an attribute reference `{authorinitials}` to the value *JB* in the current document.

To delete (undefine) an attribute use the following syntax:

```
:<name>!:
```

AttributeEntry properties

- The attribute entry line begins with colon — no white space allowed in left margin.
- *AsciiDoc* converts the `<name>` to a legal attribute name (lower case, alphanumeric and dash characters only — all other characters deleted). This allows more reader friendly text to be used.
- Leading and trailing white space is stripped from the `<value>`.
- If the `<value>` is blank then the corresponding attribute value is set to an empty string.
- Special characters in the entry `<value>` are substituted. To include special characters use the predefined `{gt}`, `{lt}`, `{amp}` attribute references.
- Attribute references contained in the entry `<value>` will be expanded.
- By default AttributeEntry values are substituted for `specialcharacters` and `attributes` (see above), if you want a different AttributeEntry substitution set the `attributeentry-subs` attribute.
- Attribute entries in the document Header are available for header markup template substitution.
- Attribute elements override configuration file and intrinsic attributes but do not override command-line attributes.

Here's another example:

```
AsciiDoc User Manual
====================
:Author:    Stuart Rackham
:Email:     srackham@methods.co.nz
:Date:      April 23, 2004
:Revision:  5.1.1
:Key words: linux, ralink, debian, wireless
:Revision history:
```

Which creates these attributes:

```
{author}, {firstname}, {lastname}, {authorinitials}, {email},
{date}, {revision}, {keywords}, {revisionhistory}
```

The preceding example is equivalent to the standard *AsciiDoc* two line document header. Actually it's a little bit different with the addition of the `{keywords}` and `{revisionhistory}` attributes [4].

Attribute Lists

An attribute list is a comma separated list of attribute values. The entire list is enclosed in square brackets. Attribute lists are used to pass parameters to macros, blocks and inline quotes.

The list consists of zero or more positional attribute values followed by zero or more named attribute

[4] The existence of a `{revisionhistory}` attribute causes a revision history file (if it exists) to be included in DocBook outputs. If a file named like `{docname}-revhistory.xml` exists in the document's directory then it will be added verbatim to the DocBook header (see the `./doc/asciidoc-revhistory.xml` example that comes with the *AsciiDoc* distribution).

values. Here are three examples:

```
[Hello]
[Bertrand Russell, The World of Mathematics (1956)]
["22 times", backcolor="#0e0e0e", options="noborders,wide"]
```

Attribute lists are evaluated as a list of Python function arguments. If this fails or any of the items do not evaluate to a string a number or None then all list items are treated as string literals.

Attribute list properties

- List attributes take precedence over existing attributes.
- List attributes can only be referenced in configuration file markup templates and tags, they are not available inside the document.
- Attribute references are allowed inside attribute lists.
- If the list contains any named attributes the all string attribute values must be quoted.
- Setting a named attribute to `None` undefines the attribute.
- Positional attributes are referred to as `{1}`,`{2}`,`{3}`,…
- Attribute `{0}` refers to the entire list (excluding the enclosing square brackets).
- If an attribute named `options` is present it is processed as a comma separated list of attributes with zero length string values. For example `[options="opt1,opt2,opt3"]` is equivalent to `[opt1="",opt2="",opt2=""]`.

Macro Attribute lists

Macros calls are suffixed with an attribute list. The list may be empty but it cannot be omitted. List entries are used to pass attribute values to macro markup templates.

AttributeList Element

An attribute list on a line by itself constitutes an *AttributeList* block element, its function is to parametrize the following block element. The list attributes are passed to the next block element for markup template substitution.

Attribute References

An attribute references is an attribute name (possibly followed by an additional parameters) enclosed in braces. When an attribute reference is encountered it is evaluated and replaced by its corresponding text value. If the attribute is undefined the line containing the attribute is dropped.

There are three types of attribute reference: *Simple*, *Conditional* and *System*.

Attribute reference behavior

- You can suppress attribute reference expansion by placing a backslash character immediately in front of the opening brace character.
- By default attribute references are not expanded in LiteralParagraphs, ListingBlocks or LiteralBlocks.

Simple Attributes References

Simple attribute references take the form `{<name>}`. If the attribute name is defined its text value is substituted otherwise the line containing the reference is dropped from the output.

Conditional Attribute References

Additional parameters are used in conjunction with the attribute name to calculate a substitution value. Conditional attribute references take the following forms:

`{<name>=<value>}`
: `<value>` is substituted if the attribute `<name>` is undefined otherwise its value is substituted. `<value>` can contain simple attribute references.

`{<name>?<value>}`
: `<value>` is substituted if the attribute `<name>` is defined otherwise an empty string is substituted. `<value>` can contain simple attribute references.

`{<name>!<value>}`
: `<value>` is substituted if the attribute `<name>` is undefined otherwise an empty string is substituted. `<value>` can contain simple attribute references.

`{<name>#<value>}`
: `<value>` is substituted if the attribute `<name>` is defined otherwise the undefined attribute entry causes the containing line to be dropped. `<value>` can contain simple attribute references.

`{<name>%<value>}`
: `<value>` is substituted if the attribute `<name>` is not defined otherwise the containing line is dropped. `<value>` can contain simple attribute references.

`{<name>@<regexp>:<value1>[:<value2>]}`
: `<value1>` is substituted if the value of attribute `<name>` matches the regular expression `<regexp>` otherwise `<value2>` is substituted. If attribute `<name>` is not defined the containing line is dropped. If `<value2>` is omitted an empty string is assumed. The values and the regular expression can contain simple attribute references. To embed colons in the values or the regular expression escape them with backslashes.

`{<name>$<regexp>:<value1>[:<value2>]}`
: Same behavior as the previous ternary attribute except for the following cases:

  `{<name>$<regexp>:<value>}`
  Substitutes `<value>` if `<name>` matches `<regexp>` otherwise the result is undefined and the containing

line is dropped.

`{<name>$<regexp>::<value>}`
: Substitutes `<value>` if `<name>` does not match `<regexp>` otherwise the result is undefined and the containing line is dropped.

Conditional attribute examples

Conditional attributes are mainly used in *AsciiDoc* configuration files — see the distribution `.conf` files for examples.

Attribute equality test
: If `{backend}` is `docbook` or `xhtml11` the example evaluates to “DocBook or XHTML backend” otherwise it evaluates to “some other backend”:

```
{backend@docbook|xhtml11:DocBook or XHTML backend:some other backend}
```

Attribute value map
: This example maps the `frame` attribute values [`topbot`, `all`, `none`, `sides`] to [`hsides`, `border`, `void`, `vsides`]:

```
{frame@topbot:hsides}{frame@all:border}{frame@none:void}{frame@sides:vsides}
```

System Attribute References

System attribute references generate the attribute text value by executing a predefined action that is parametrized by a single argument. The syntax is `{<action>:<argument>}`.

`{eval:<expression>}`
: Substitutes the result of the Python `<expression>`. If `<expression>` evaluates to `None` or `False` the reference is deemed undefined and the line containing the reference is dropped from the output. If the expression evaluates to `True` the attribute evaluates to an empty string. In all remaining cases the attribute evaluates to a string representation of the `<expression>` result.

`{include:<filename>}`
: Substitutes contents of the file named `<filename>`.

- The included file is read at the time of attribute substitution.
- If the file does not exist a warning is emitted and the line containing the reference is dropped from the output file.
- Tabs are expanded based on the current *tabsize* attribute value.

`{sys:<command>}`
: Substitutes the stdout generated by the execution of the shell `<command>`.

`{sys2:<command>}`
: Substitutes the stdout and stderr generated by the execution of the shell `<command>`.

System reference behavior

- System attribute arguments can contain non-system attribute references.
- Closing brace characters inside system attribute arguments must be escaped them with a backslash.

Intrinsic Attributes

Intrinsic attributes are simple attributes that are created automatically from *AsciiDoc* document header parameters, `asciidoc(1)` command-line arguments, execution parameters along with attributes defined in the default configuration files. Here's the list of predefined intrinsic attributes:

```
{asciidoc-dir}         the asciidoc(1) application directory
{asciidoc-version}     the version of asciidoc(1)
{author}               author's full name
{authored}             empty string '' if {author} or {email} defined,
{authorinitials}       author initials (from document header)
{backend-<backend>}    empty string ''
{<backend>-<doctype>}  empty string ''
{backend}              document backend specified by `-b` option
{basebackend-<base>}   empty string ''
{basebackend}          html or docbook
{brvbar}               broken vertical bar (|) character
{date}                 document date (from document header)
{docname}              document file name without extension
{doctitle}             document title (from document header)
{doctype-<doctype>}    empty string ''
{doctype}              document type specified by `-d` option
{email}                author's email address (from document header)
{empty}                empty string ''
{filetype-<fileext>}   empty string ''
{filetype}             output file name file extension
{firstname}            author first name (from document header)
{gt}                   greater than (>) character
{id}                   running block id generated by BlockId elements
{indir}                document input directory name (note 1)
{infile}               input file name (note 1)
{lastname}             author last name (from document header)
{listindex}            the list index (1..) of the most recent list item
{localdate}            the current date
{localtime}            the current time
{lt}                   less than (<) character
{manname}              manpage name (defined in NAME section)
{manpurpose}           manpage (defined in NAME section)
{mantitle}             document title minus the manpage volume number
{manvolnum}            manpage volume number (1..8) (from document header)
{middlename}           author middle name (from document header)
{outdir}               document output directory name (note 1)
{outfile}              output file name (note 1)
{revision}             document revision number (from document header)
{sectnum}              section number (in section titles)
{title}                section title (in titled elements)
{user-dir}             the ~/.asciidoc directory (if it exists)
{verbose}              defined as '' if --verbose command option specified
```

NOTES

1. Intrinsic attributes are global so avoid defining custom attributes with the same names.

2. `{infile}`, `{outdir}`, `{infile}`, `{indir}` attributes are effectively read-only (you can set them but it won't affect the input or output file paths).

3. See also the xhtml11 subsection for attributes that relate to *AsciiDoc* XHTML file generation.

4. The entries that translate to blank strings are designed to be used for conditional text inclusion. You can also use the `ifdef`, `ifndef` and `endif` System macros for conditional inclusion. [5]

Block Element Definitions

The syntax and behavior of Paragraph, DelimitedBlock, List and Table block elements is determined by block definitions contained in *AsciiDoc* configuration file sections.

Each definition consists of a section title followed by one or more section entries. Each entry defines a block parameter controlling some aspect of the block's behavior. Here's an example:

```
[blockdef-listing]
delimiter=^-{4,}$
template=listingblock
presubs=specialcharacters,callouts
```

AsciiDoc Paragraph, DelimitedBlock, List and Table block elements share a common subset of configuration file parameters:

delimiter
: A Python regular expression that matches the first line of a block element — in the case of DelimitedBlocks it also matches the last line. Table elements don't have an explicit delimiter — they synthesize their delimiters at runtime.

template
: The name of the configuration file markup template section that will envelope the block contents. The pipe | character is substituted for the block contents. List elements use a set of (list specific) tag parameters instead of a single template.

options
: A comma delimited list of element specific option names.

subs, presubs, postsubs

- *presubs* and *postsubs* are lists of comma separated substitutions that are performed on the block contents. *presubs* is applied first, *postsubs* (if specified) second.

- *subs* is an alias for *presubs*.

- If a *filter* is allowed (Paragraphs and DelimitedBlocks) and has been specified then *presubs* and *postsubs* substitutions are performed before and after the filter is run respectively.

- Allowed values: *specialcharacters, quotes, specialwords, replacements, macros, attributes, callouts*.

[5] Conditional inclusion using `ifdef` and `ifndef` macros differs from attribute conditional inclusion in that the former occurs when the file is read while the latter occurs when the contents are written.

- The following composite values are also allowed:

  none
  No substitutions.

  normal
  The following substitutions: *specialcharacters*,*quotes*,*attributes*,*specialwords*, *replacements*,*macros*,*passthroughs*.

  verbatim
  specialcharacters and *callouts* substitutions.

- *normal* and *verbatim* substitutions can be redefined by with `subsnormal` and `subsverbatim` entries in a configuration file `[misc]` section.
- The substitutions are processed in the order in which they are listed and can appear more than once.

filter
: This optional entry specifies an executable shell command for processing block content (Paragraphs and DelimitedBlocks). The filter command can contain attribute references.

posattrs
: Optional comma separated list of positional attribute names. This list maps positional attributes (in the block's attribute list) to named block attributes. The following example, from the QuoteBlock definition, maps the first and section positional attributes:

```
posattrs=attribution,citetitle
```

style
: This optional parameter specifies the default style name.

<stylename>-style
: Optional style definition (see Styles below).

The following block parameters behave like document attributes and can be set in block attribute lists and style definitions: *template*, *options*, *subs*, *presubs*, *postsubs*, *filter*.

Styles

A style is a set of block attributes bundled as a single named attribute. The following example defines a style named *verbatim*:

```
verbatim-style=template="literalblock",subs="verbatim",font="monospaced"
```

All style parameter names must be suffixed with `-style` and the style parameter value is in the form of a list of named attributes.

Paragraphs

Paragraph translation is controlled by `[paradef*]` configuration file section entries. Users can define new types of paragraphs and modify the behavior of existing types by editing *AsciiDoc* configuration files.

Here is the shipped Default paragraph definition:

```
[paradef-default]
delimiter=(?P<text>\S.*)
template=paragraph
```

The Default paragraph definition has a couple of special properties:

1. It must exist and be defined in a configuration file section named `[paradef-default]`.
2. Irrespective of its position in the configuration files default paragraph document matches are attempted only after trying all other paragraph types.

Paragraph specific block parameter notes:

delimiter
: This regular expression must contain the named group *text* which matches the text on the first line. Paragraphs are terminated by a blank line, the end of file, or the start of a DelimitedBlock.

options
: The only allowable option is *listelement*. The *listelement* option specifies that paragraphs of this type will automatically be considered part of immediately preceding list items.

Paragraph processing proceeds as follows:

1. The paragraph text is aligned to the left margin.
2. Optional *presubs* inline substitutions are performed on the paragraph text.
3. If a filter command is specified it is executed and the paragraph text piped to its standard input; the filter output replaces the paragraph text.
4. Optional *postsubs* inline substitutions are performed on the paragraph text.
5. The paragraph text is enveloped by the paragraph's markup template and written to the output file.

Delimited Blocks

DelimitedBlock specific block definition notes:

options
: Allowed values are:

  sectionbody
  The block contents are processed as a SectionBody.

skip
The block is treated as a comment (see *CommentBlocks*).

list
The block is a list block.

presubs, *postsubs* and *filter* entries are meaningless when *sectionbody*, *skip* or *list* options are set.

DelimitedBlock processing proceeds as follows:

1. Optional *presubs* substitutions are performed on the block contents.
2. If a filter is specified it is executed and the block's contents piped to its standard input. The filter output replaces the block contents.
3. Optional *postsubs* substitutions are performed on the block contents.
4. The block contents is enveloped by the block's markup template and written to the output file.

Attribute expansion is performed on the block filter command before it is executed, this is useful for passing arguments to the filter.

Lists

List behavior and syntax is determined by `[listdef*]` configuration file sections. The user can change existing list behavior and add new list types by editing configuration files.

List specific block definition notes:

type
This is either *bulleted*,*numbered*,*labeled* or *callout*.

delimiter
A Python regular expression that matches the first line of a list element entry. This expression must contain the named group *text* which matches text in the first line.

subs
Substitutions that are performed on list item text and terms.

listtag
The name of the tag that envelopes the List.

itemtag
The name of the tag that envelopes the ListItem.

texttag
The name of the tag that envelopes the list item text.

labeltag
: The name of the tag that envelopes a variable list label.

entrytag
: The name of the tag that envelopes a labeled list entry.

The tag entries map the *AsciiDoc* list structure to backend markup; see the *AsciiDoc* distribution `.conf` configuration files for examples.

Tables

Table behavior and syntax is determined by `[tabledef*]` configuration file sections. The user can change existing list behavior and add new list types by editing configuration files.

Table specific block definition notes:

fillchar
: A single character that fills table ruler and underline lines.

subs
: Substitutions performed on table data items.

format
: The source row data format (*fixed*, *csv* or *dsv*).

comspec
: The table *comspec* tag definition.

headrow, footrow, bodyrow
: Table header, footer and body row tag definitions. *headrow* and *footrow* table definition entries default to *bodyrow* if they are undefined.

headdata, footdata, bodydata
: Table header, footer and body data tag definitions. *headdata* and *footdata* table definition entries default to *bodydata* if they are undefined.

Table behavior is also influenced by the following `[miscellaneous]` configuration file entries:

textwidth
: The page width (in characters) of the source text. This setting is compared to the the table ruler width when calculating the relative size of character ruler tables on the output page.

pagewidth
: This integer value is the printable width of the output media. Used to calculate *colwidth* and *tablewidth* substitution values.

pageunits
: The units of width in output markup width attribute values.

Table definition behavior

- The output markup generation is specifically designed to work with the HTML and CALS (DocBook) table models, but should be adaptable to most XML table schema.
- Table definitions can be “mixed in” from multiple cascading configuration files.
- New table definitions inherit the default table definition (*[tabledef-default]*) so you only need to override those conf file entries that require modification when defining a new table type.

Filters

Filters are external shell commands used to process Paragraph and DelimitedBlock content; they are specified in configuration file Paragraph and DelimitedBlock definitions.

There's nothing special about the filters, they're just standard UNIX filters: they read text from the standard input, process it, and write to the standard output.

Attribute substitution is performed on the filter command prior to execution — attributes can be used to pass parameters from the *AsciiDoc* source document to the filter.

Filters can potentially generate unsafe output. Before installing a filter you should verify that it can't be coerced into generating malicious output or exposing sensitive information.

Filter functionality is currently only available on POSIX platforms (this includes Cygwin).

Filter Search Paths

If the filter command does not specify a directory path then `asciidoc(1)` searches for the command:

- First it looks in the user's `$HOME/.asciidoc/filters` directory.
- Next the `/etc/asciidoc/filters` directory is searched.
- Then it looks in the `asciidoc(1) ./filters` directory.
- Finally it relies on the executing shell to search the environment search path (`$PATH`).

Filter Configuration Files

Filters are normally accompanied by a configuration file containing a filter Paragraph or filter DelimitedBlock definition and corresponding markup templates.

By convention delimiters belonging to DelimitedBlock filters distributed with *AsciiDoc* consist of a word (normally a noun identifying the block content) followed by four or more tilde characters.

`asciidoc(1)` auto-loads all `.conf` files found in the filter search paths (see previous section).

Code Filter

AsciiDoc comes with a simple minded for highlighting source code keywords and comments. See also the `./filters/code-filter-readme.txt` file.

This filter primarily to demonstrate how to write a filter — it's much to simplistic to be passed off as a code syntax highlighter. If you want a full featured multi-language highlighter use the Source Code Highlighter Filter.

```
.Code filter example
[python]
code~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
''' A multi-line
    comment.'''
def sub_word(mo):
    ''' Single line comment.'''
    word = mo.group('word')   # Inline comment
    if word in keywords[language]:
        return quote + word + quote
    else:
        return word
code~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Outputs:

Example 5. Code filter example

```
''' A multi-line
    comment.'''
def sub_word(mo):
    ''' Single line comment.'''
    word = mo.group('word')     # Inline comment
    if word in keywords[language]:
        return quote + word + quote
    else:
        return word
```

Source Code Highlighter Filter

A source code highlighter filter [http://www.methods.co.nz/asciidoc/source-highlight-filter.html] can be found in the *AsciiDoc* distribution `./filters` directory. It uses GNU source-highlight [http://www.gnu.org/software/src-highlite/] to generate nicely formatted source code for most common programming languages.

Music Filter

A music filter [http://www.methods.co.nz/asciidoc/music-filter.html] is included in the distribution `./filters` directory. It translates music in LilyPond [http://lilypond.org/] or ABC [http://abcnotation.org.uk/] notation to standard Western classical notation in the form of a trimmed PNG image which is automatically inserted into the output document.

Converting DocBook to other file formats

DocBook files are validated, parsed and translated by a combination of applications collectively called a DocBook *tool chain*. The function of a tool chain is to read the DocBook markup (produced by *AsciiDoc*) and transform it to a presentation format (for example HTML, PDF, HTML Help).

A wide range of user output format requirements coupled with a choice of available tools and stylesheets results in many valid tool chain combinations.

The DocBook toolchain currently used for processing *AsciiDoc* documentation is xsltproc(1), FOP and DocBook XSL Stylesheets. These tools are freely available for Linux and Windows systems.

Why Generate HTML via DocBook?

AsciiDoc produces nicely styled HTML directly without requiring a DocBook toolchain but there are also advantages in going the DocBook route:

- HTML from DocBook includes automatically generated indexes, tables of contents, footnotes, lists of figures and tables.
- DocBook toolchains can also (optionally) generate separate (chunked) linked HTML pages for each document section.
- Toolchain processing performs link and document validity checks.
- If the DocBook *lang* attribute is set then things like table of contents, revision history, figure and table captions and admonition captions will be output in the specified language (setting the *AsciiDoc lang* attribute sets the DocBook *lang* attribute).

On the other hand, HTML output directly from *AsciiDoc* is much faster, is easily customized and can be used in situations where there is no suitable DocBook toolchain (see the *AsciiDoc* website [http://www.methods.co.nz/asciidoc/] for example).

If you require output formats other than HTML you would feed *AsciiDoc*'s DocBook output to a DocBook toolchain. The distributed PDF documents have been generated in this way.

The toolchain processing steps are:

1. Convert *AsciiDoc* (`*.txt`) documents to DocBook XML (`*.xml`) using *AsciiDoc*.
2. Convert DocBook XML documents to HTML, XSL-FO or HTML Help files using DocBook XSL Stylesheets and the `xsltproc(1)` XML parser.

3. Convert the XSL-FO (`*.fo`) files to PDF using FOP and HTML Help source (`*.hhp`) files to HTML Help (`*.chm`) files using the Microsoft HTML Help Compiler.

These steps can be automated by using the *AsciiDoc* `a2x(1)` toolchain wrapper command.

Lazy DocBook Conversion

Depending on your Linux distribution toolchain installation can be a mission for users used to a GUI environment, even more so under Microsoft Windows. So you may like to try the XMLmind FO Converter [http://www.xmlmind.com/foconverter/], it contains a GUI *XSL Utility* which makes a creditable job of converting *AsciiDoc* generated DocBook files to RTF, HTML and Open Document formats. The *FO Converter Personal Edition* can be used free of charge and a Windows installer is available and there is also a UNIX version. Thanks to Matthew Marshall for this tip.

a2x Toolchain Wrapper

One of the biggest hurdles for new users seems to be using a DocBook XML toolchain. `a2x(1)` can help — it's a toolchain wrapper command that will generate XHTML (chunked and unchunked), PDF, man page, HTML Help and text file outputs from an *AsciiDoc* text file. `a2x(1)` does all the grunt work associated with generating and sequencing the toolchain commands and managing intermediate and output files. `a2x(1)` also optionally deploys admonition and navigation icons and a CSS stylesheet. See the `a2x(1)` man page for more details. All you need is xsltproc(1), DocBook XSL Stylesheets and optionally FOP (if you want PDF) or lynx(1) (if you want text).

The following example generates `doc/quickstart.pdf` from the *AsciiDoc* `doc/quickstart.txt` source file:

```
$ a2x -f pdf --icons doc/quickstart.txt
```

See the `a2x(1)` man page for details.

Use the –`verbose` command-line option to view executed toolchain commands.

Toolchain Components

AsciiDoc
: Converts *AsciiDoc* (`*.txt`) files to DocBook XML (`*.xml`) files.

DocBook XSL Stylesheets [http://docbook.sourceforge.net/projects/xsl/]
: These are a set of XSL stylesheets containing rules for converting DocBook XML documents to HTML, XSL-FO, manpage and HTML Help files. The stylesheets are used in conjunction with an XML parser such as `xsltproc(1)`.

xsltproc [http://www.xmlsoft.org]
: `xsltproc` is a command line XML parser for applying XSLT stylesheets (in our case the DocBook XSL Stylesheets) to XML documents.

FOP
: The Apache Formatting Objects Processor converts XSL-FO (`*.fo`) files to PDF files (see the FOP section).

Microsoft Help Compiler
: The Microsoft HTML Help Compiler (`hhc.exe`) is a command-line tool that converts HTML Help source files to a single HTML Help (`*.chm`) file. It runs on MS Windows platforms and can be downloaded from http://www.microsoft.com.

AsciiDoc DocBook XSL Drivers

You will have noticed that the distributed PDF, HTML and HTML Help documentation files (for example `./doc/asciidoc.html`) are not the plain outputs produced using the default DocBook XSL Stylesheets configuration. This is because they have been processed using customized DocBook XSL Stylesheet drivers along with (in the case of HTML outputs) the custom `./stylesheets/docbook.css` CSS stylesheet.

You'll find the customized DocBook XSL drivers along with additional documentation in the distribution `./docbook-xsl` directory. The examples that follow are executed from the distribution documentation (`./doc`) directory.

`common.xsl`
: Shared driver parameters. This file is not used directly but is included in all the following drivers.

`chunked.xsl`
: Generate chunked XHTML (separate HTML pages for each document section) in the `./doc/chunked` directory. For example:

```
$ python ../asciidoc.py -b docbook asciidoc.txt
$ xsltproc --nonet ../docbook-xsl/chunked.xsl asciidoc.xml
```

`fo.xsl`
: Generate XSL Formatting Object (`*.fo`) files for subsequent PDF file generation using FOP. For example:

```
$ python ../asciidoc.py -b docbook article.txt
$ xsltproc --nonet ../docbook-xsl/fo.xsl article.xml > article.fo
$ fop.sh article.fo article.pdf
```

`htmlhelp.xsl`
: Generate Microsoft HTML Help source files for the MS HTML Help Compiler in the `./doc/htmlhelp` directory. This example is run on MS Windows from a Cygwin shell prompt:

```
$ python ../asciidoc.py -b docbook asciidoc.txt
$ xsltproc --nonet ../docbook-xsl/htmlhelp.xsl asciidoc.xml
$ c:/Program\ Files/HTML\ Help\ Workshop/hhc.exe htmlhelp.hhp
```

`manpage.xsl`
: Generate a `roff(1)` format UNIX man page from a DocBook XML *refentry* document. This example generates an `asciidoc.1` man page file:

```
$ python ../asciidoc.py -d manpage -b docbook asciidoc.1.txt
$ xsltproc --nonet ../docbook-xsl/manpage.xsl asciidoc.1.xml
```

`xhtml.xsl`
: Convert a DocBook XML file to a single XHTML file. For example:

```
$ python ../asciidoc.py -b docbook asciidoc.txt
$ xsltproc --nonet ../docbook-xsl/xhtml.xsl asciidoc.xml > asciidoc.html
```

If you want to see how the complete documentation set is processed take a look at the A-A-P script `./doc/main.aap`.

FOP

XSL Stylesheets can be used to generate FO (Formatting Object) files, which in turn can be used to produce PDF files using the Apache Formatting Object Processor program (FOP). The FOP home page is at http://xml.apache.org/fop/.

As of version 0.20.5 installation and configuration of FOP is a manual process. You also need a working Java Runtime to run FOP. You'll find FOP and Java installation information in the appendices.

> Once you've got FOP installed use the *AsciiDoc* `a2x(1)` toolchain wrapper to generate PDF files from *AsciiDoc* source.

Generating Plain Text Files

AsciiDoc does not have a text backend (for most purposes *AsciiDoc* source text is fine), however you can convert *AsciiDoc* text files to formatted text using the *AsciiDoc* `a2x(1)` toolchain wrapper utility.

XML and Character Sets

The default XML character set `UTF-8` is used when *AsciiDoc* generates DocBook files but this can be changed by setting the `xmldecl` entry in the `[attributes]` section of the `docbook.conf` file or by composing your own configuration file `[header]` section).

> If you get an *undefined entity* error when processing DocBook files you'll may find that you've used an undefined HTML character entity. An easy (although inelegant) fix is to use the character's character code instead of its symbolic name (for example use `&#160;` instead of `&nbsp;`).

If your system has been configured with an XML catalog you may find a number of entity sets are already automatically included.

PDF Fonts

The Adobe PDF Specification states that the following 14 fonts should be available to every PDF reader: Helvetica (normal, bold, italic, bold italic), Times (normal, bold, italic, bold italic), Courier (normal, bold, italic, bold italic), Symbol and ZapfDingbats. Non-standard fonts should be embedded in the distributed document.

Help Commands

The `asciidoc(1)` command has a `--help` option which prints help topics to stdout. The default topic summarizes `asciidoc(1)` usage:

```
$ asciidoc --help
```

To print a list of help topics:

```
$ asciidoc --help=topics
```

To print a help topic specify the topic name as a command argument. Help topic names can be shortened so long as they are not ambiguous. Examples:

```
$ asciidoc --help=manpage
$ asciidoc -hm              # Short version of previous example.
$ asciidoc --help=syntax
$ asciidoc -hs              # Short version of previous example.
```

Customizing Help

To change, delete or add your own help topics edit a help configuration file. The help file name `help-<lang>.conf` is based on the setting of the `lang` attribute, it defaults to `help.conf` (English). The help file location will depend on whether you want the topics to apply to all users or just the current user.

The help topic files have the same named section format as other configuration files. The `help.conf` files are stored in the same locations and loaded in the same order as other configuration files.

When the `--help` command-line option is specified *AsciiDoc* loads the appropriate help files and then prints the contents of the section whose name matches the help topic name. If a topic name is not specified `default` is used. You don't need to specify the whole help topic name on the command-line, just enough letters to ensure it's not ambiguous. If a matching help file section is not found a list of available topics is printed.

Tips and Tricks

Know Your Editor

Writing *AsciiDoc* documents will be a whole lot more pleasant if you know your favorite text editor. Learn how to indent and reformat text blocks, paragraphs, lists and sentences. Tips for *vim* users follow.

Vim Commands for Formatting AsciiDoc

Text Wrap Paragraphs

Use the vim `:gq` command to reformat paragraphs. Setting the *textwidth* sets the right text wrap margin; for example:

```
:set textwidth=70
```

To reformat a paragraph:

1. Position the cursor at the start of the paragraph.
2. Type `gq}`.

Execute `:help gq` command to read about the vim gq command.

- Assign the `gq}` command to the Q key with the `nnoremap Q gq}` command or put it in your `~/.vimrc` file to so it's always available (see the Example ~/.vimrc file).
- Put `set` commands in your `~/.vimrc` file so you don't have to enter them manually Example ~/.vimrc file).
- The Vim website (http://www.vim.org) has a wealth of resources, including scripts for automated spell checking and ASCII Art drawing.

Format Lists

The `gq` command can also be used to format bulleted and numbered lists. First you need to set the `comments` and `formatoptions` (see the Example ~/.vimrc file).

Now you can format simple lists that use dash, asterisk, period and plus bullets along with numbered ordered lists:

1. Position the cursor at the start of the list.
2. Type `gq}`.

Indent Paragraphs

Indent whole paragraphs by indenting the fist line with the desired indent and then executing the `gq}` command.

Example ~/.vimrc File

```
" Show tabs and trailing characters.
set listchars=tab:»·,trail:·
set list
```

```
" Don't highlight searched text.
highlight clear Search

" Don't move to matched text while search pattern is being entered.
set noincsearch

" Q command to reformat paragraphs and list.
nnoremap Q gq}

" W command to delete trailing white space and Dos-returns and to expand tabs
" to spaces.
nnoremap W :%s/[\r \t]\+$//<CR>:set et<CR>:retab!<CR>

autocmd BufRead,BufNewFile *.txt,README,TODO,CHANGELOG,NOTES
        \ setlocal autoindent expandtab tabstop=8 softtabstop=2 shiftwidth=2
        \ textwidth=70 wrap formatoptions=tcqn
        \ comments=s1:/*,ex:*/,://,b:#,:%,:XCOMM,fb:-,fb:*,fb:+,fb:.,fb:>
```

Troubleshooting

- The `asciidoc(1)` `-v` (`–verbose`) command-line option displays the order of configuration file loading and warns of potential configuration file problems.
- Not all valid *AsciiDoc* documents produce valid backend markup. Read the *AsciiDoc* Backends section if *AsciiDoc* output is rejected as non-conformant by a backend processor.

Gotchas

Incorrect character encoding

If you get an error message like `'UTF-8' codec can't decode …` then you source file contains invalid UTF-8 characters — set the *AsciiDoc* encoding attribute for the correct character set (typically ISO-8859-1 (Latin-1) for European languages).

Misinterpreted text formatting

If text in your document is incorrectly interpreted as formatting instructions you can suppress formatting by placing a backslash character immediately in front of the leading quote character(s). For example in the following line the backslash prevents text between the two asterisks from being output in a strong (bold) font:

```
Add `\*.cs` files and `*.resx` files.
```

Overlapping text formatting

Overlapping text formatting will generate illegal overlapping markup tags which will result in downstream XML parsing errors. Here's an example:

```
Some *strong markup _that overlaps* emphasized markup_.
```

Ambiguous underlines

A DelimitedBlock can immediately follow paragraph without an intervening blank line, but be careful, a single line paragraph underline may be misinterpreted as a section title underline resulting in a “closing block delimiter expected” error.

Ambiguous ordered list items

Lines beginning with numbers at the end of sentences will be interpreted as ordered list items. The following example (incorrectly) begins a new list with item number 1999:

```
He was last sighted in
1999. Since then things have moved on.
```

The *list item out of sequence* warning makes it unlikely that this problem will go unnoticed.

Escaping inside DSV table data

Delimiter separated text uses C style backslash escape sequences. If you want to enter a backslash (for example, to escape *AsciiDoc* text formatting or an inline macro) you need to escape it by entering two backslashes.

Special characters in attribute values

Special character substitution precedes attribute substitution so if attribute values contain special characters you may, depending on the substitution context, need to escape the special characters yourself. For example:

```
$ asciidoc -a 'companyname=Bill &amp; Ben' mydoc.txt
```

Macro attribute lists

If named attribute list entries are present then all string attribute values must be quoted. For example:

```
["Desktop screenshot",width=32]
```

Combining Separate Documents

You have a number of stand-alone *AsciiDoc* documents that you want to process as a single document. Simply processing them with a series of `include` macros won't work, because instead of starting at level 1 the section levels of the combined document start at level 0 (the document title level).

The solution is to redefine the title underlines so that document and section titles are pushed down one level.

1. Push the standard title underlines down one level by defining a new level 0 underline in a custom configuration file. For example `combined.conf`:

   ```
   [titles]
   underlines="__","==","--","~~","^^"
   ```

2. If you use single line titles you'll need to make corresponding adjustments to the `[titles]` section `sect0`…`sect4` entries.

3. Create a top level wrapper document. For example `combined.txt`:

   ```
   Combined Document Title
   _______________________

   include::document1.txt[]

   include::document2.txt[]
   ```

```
include::document3.txt[]
```

4. Process the wrapper document. For example:

```
$ asciidoc --conf-file=combined.conf combined.txt
```

Actually the –`conf-file` option is unnecessary as `asciidoc(1)` automatically looks for a same-named `.conf` file.

- The combined document title uses the newly defined level 0 underline (underscore characters).
- Put a blank line between the `include` macro lines to ensure the title of the included document is not seen as part of the last paragraph of the previous document.
- You won't want document Headers (Author and Revision lines) in the included files — conditionally exclude them if they are necessary for stand-alone processing.

Processing Document Sections Separately

You have divided your *AsciiDoc* document into separate files (one per top level section) which are combined and processed with the following top level document:

```
Combined Document Title
=======================
Joe Bloggs
v1.0, 12-Aug-03

include::section1.txt[]

include::section2.txt[]

include::section3.txt[]
```

You also want to process the section files as separate documents. This is easy because `asciidoc(1)` will quite happily process `section1.txt`, `section2.txt` and `section3.txt` separately.

If you want to promote the section levels up one level, so the document is processed just like a stand-alone document, then pop the section underline definition up one level:

```
[titles]
underlines="--","~~","^^","++","__"
```

The last `"__"` underline is a dummy that won't actually be used but is necessary to legitimize the underline definition.

This is just the reverse of the technique used for combining separate documents explained in the previous section.

Processing Document Chunks

`asciidoc(1)` can be used as a filter, so you can pipe chunks of text through it. For example:

```
$ echo 'Hello *World!*' | asciidoc -s -
<div class="para"><p>Hello <strong>World!</strong></p></div>
```

The `-s` (–`no-header-footer`) command-line option suppresses header and footer output and is useful if the processed output is to be included in another file.

Badges in HTML Page Footers

See the `[footer]` section in the *AsciiDoc* distribution `xhtml11.conf` configuration file.

Pretty Printing AsciiDoc Output

If the indentation and layout of the `asciidoc(1)` output is not to your liking you can:

1. Change the indentation and layout of configuration file markup template sections. The `{empty}` glossary entry is useful for outputting trailing blank lines in markup templates.
2. Or use Dave Raggett's excellent *HTML Tidy* program to tidy `asciidoc(1)` output. Example:

   ```
   $ asciidoc -b docbook -o - mydoc.txt | tidy -indent -xml >mydoc.xml
   ```

HTML Tidy can be downloaded from http://tidy.sourceforge.net/

Supporting Minor DocBook DTD Variations

The conditional inclusion of DocBook SGML markup at the end of the distribution `docbook.conf` file illustrates how to support minor DTD variations. The included sections override corresponding entries from preceding sections.

Shipping Stand-alone AsciiDoc Source

Reproducing presentation documents from someone else's source has one major problem: unless your configuration files are the same as the creator's you won't get the same output.

The solution is to create a single backend specific configuration file using the `asciidoc(1)` `-c` (–`dump-conf`) command-line option. You then ship this file along with the *AsciiDoc* source document plus the `asciidoc.py` script. The only end user requirement is that they have Python installed (and of course that they consider you a trusted source). This example creates a composite HTML configuration file for `mydoc.txt`:

```
$ asciidoc -cb xhtml11 mydoc.txt > mydoc-xhtml11.conf
```

Ship `mydoc.txt`, `mydoc-html.conf`, and `asciidoc.py`. With these three files (and a Python interpreter) the recipient can regenerate the HMTL output:

```
$ ./asciidoc.py -eb xhtml11 mydoc.txt
```

The `-e` (–`no-conf`) option excludes the use of implicit configuration files, ensuring that only entries from the `mydoc-html.conf` configuration are used.

Inserting Blank Space

Adjust your style sheets to add the correct separation between block elements. Inserting blank paragraphs containing a single non-breaking space character `{nbsp}` works but is an ad hoc solution compared to using style sheets.

Closing Open Sections

You can close off section tags up to level `N` by calling the `eval::[Section.setlevel(N)]` system macro. This is useful if you want to include a section composed of raw markup. The following example includes a DocBook glossary division at the top section level (level 0):

```
ifdef::backend-docbook[]

eval::[Section.setlevel(0)]

++++++++++++++++++++++++++++++
<glossary>
  <title>Glossary</title>
  <glossdiv>
  ...
  </glossdiv>
</glossary>
++++++++++++++++++++++++++++++
endif::backend-docbook[]
```

Validating Output Files

Use `xmllint(1)` to check the *AsciiDoc* generated markup is both well formed and valid. Here are some examples:

```
$ xmllint --nonet --noout --valid docbook-file.xml
$ xmllint --nonet --noout --valid xhtml11-file.html
$ xmllint --nonet --noout --valid --html html4-file.html
```

The –`valid` option checks the file is valid against the document type's DTD, if the DTD is not installed in your system's catalog then it will be fetched from its Internet location. If you omit the –`valid` option the document will only be checked that it is well formed.

Glossary

Block element
: An *AsciiDoc* block element is a document entity composed of one or more whole lines of text.

Inline element
: *AsciiDoc* inline elements occur within block element textual content, they perform formatting and substitution tasks.

Formal element
: An *AsciiDoc* block element that has a BlockTitle. Formal elements are normally listed in front or back matter, for example lists of tables, examples

and figures.

Verbatim element
: The word verbatim indicates that white space and line breaks in the source document are to be preserved in the output document.

A. Migration Notes

Version 7 to version 8

- A new set of quotes has been introduced which may match inline text in existing documents — if they do you'll need to escape the matched text with backslashes.
- The index entry inline macro syntax has changed — if your documents include indexes you may need to edit them.
- Replaced `a2x(1)` `--no-icons` and `--no-copy` options with their negated equivalents: `--icons` and `--copy` respectively. The default behavior has also changed — the use of icons and copying of icon and CSS files must be specified explicitly with the `--icons` and `--copy` options.

The rationale for the changes can be found in the *AsciiDoc* `CHANGELOG`.

If you want to disable unconstrained quotes, the new alternative constrained quotes syntax and the new index entry syntax then you can define the attribute `asciidoc7compatible` (for example by using the `-a asciidoc7compatible` command-line option).

Version 6 to version 7

The changes that affect the most users relate to renamed and deprecated backends and command-line syntax:

1. The *html* backend has been renamed *html4*.
2. The *xhtml* backend has been deprecated to *xhtml-deprecated* (use the new *xhtml11* backend in preference).
3. The use of CSS specific `css` and `css-embedded` backends has been dropped in favor of using attributes (see the table below and xhtml backend attributes).
4. Deprecated features that emitted warnings in prior versions are no longer tolerated.
5. The command-line syntax for deleting (undefining) an attribute has changed from `-a ^name` to `-a name!`.

Table A.1. Equivalent command-line syntax

Version 6 (old)	**Version 7 (new)**	**Version 7 (backward compatible)**
-b html	-b html4	-b html4
-b css	-b xhtml11 -a linkcss -a icons	-b xhtml-deprecated -a css -a linkcss -a icons
-b css-embedded	-b xhtml11 -a icons	-b xhtml-deprecated -a css -a icons
-b xhtml	-b xhtml11	-b xhtml-deprecated
-b docbook-sgml	-b docbook -a sgml	-b docbook -a sgml

If you've customized version 6 distribution stylesheets then you'll need to either bring them in line with the new `./stylesheets/xhtml11*.css` class and id names or stick with the backward compatible `xhtml-deprecated` backend.

Changes to configuration file syntax:

1. To undefine an attribute in the `[attributes]` section use `name!` instead of `name` (`name` now sets that attribute to a blank string).

B. Packager Notes

Read the `README` and `INSTALL` files (in the distribution root directory) for install prerequisites and procedures.

The distribution `install.sh` shell script is the canonical installation procedure and is the definitive installation description. Here's a summary of the installation procedure:

- Unpack entire distribution tarball to `/usr/share/asciidoc/`.
- Move `asciidoc.py` to `/usr/bin/`; rename to `asciidoc`; if necessary modify shebang line; ensure executable permissions are set.
- Move `a2x` to `/usr/bin/`; if necessary modify shebang line; ensure executable permissions are set.
- Move the `./*.conf` files to `/etc/asciidoc/`.
- Move `./filters/{*.conf,*.py}` to `/etc/asciidoc/filters/`.
- Move `./docbook-xsl/*.xsl` to `/etc/asciidoc/docbook-xsl/`.
- Copy `./stylesheets/*.css` to `/etc/asciidoc/stylesheets/`.
- Copy `./javascripts/*.js` to `/etc/asciidoc/javascripts/`.
- Copy `./images/icons/*` to `/etc/asciidoc/images/icons/` (recursively including the `icons` subdirectory and its contents).

- Compress the `asciidoc(1)` and ax2(1) man pages (`./doc/*.1`) with gzip(1) and move them to `/usr/share/man/man1/`.
- If Vim is installed then install Vim syntax and filetype detection files.

Leaving stylesheets and images in `/usr/share/asciidoc/` ensures the docs and example website are not broken.

C. AsciiDoc Safe Mode

AsciiDoc safe mode skips potentially dangerous sections in *AsciiDoc* source files by inhibiting the execution of arbitrary code or the inclusion of arbitrary files.

The safe mode is enabled by default and can only be disabled using the `asciidoc(1)` `–unsafe` command-line option.

Safe mode constraints

- `eval`, `sys` and `sys2` executable attributes and block macros are not executed.
- `include::<filename>[]` and `include1::<filename>[]` block macro files must reside inside the parent file's directory.
- `{include:<filename>}` executable attribute files must reside inside the source document directory.
- Passthrough Blocks are dropped.

The safe mode is not designed to protect against unsafe *AsciiDoc* configuration files. Be especially careful when:

1. Implementing filters.
2. Implementing elements that don't escape special characters.
3. Accepting configuration files from untrusted sources.

D. Installing FOP on Windows

1. Download latest FOP distribution from http://xml.apache.org/fop/.
2. Unzip to `C:\bin`.

3. Edit the distribution `fop.bat` file and put it in the search `PATH`:

   ```
   set LOCAL_FOP_HOME=C:\bin\fop-0.20.5\
   ```

4. Download the JIMI image processing library from http://java.sun.com/products/jimi/.

5. Extract the `JimiProClasses.jar` library from the JIMI distribution and copy to the FOP `./lib` directory.

6. Edit the distribution `fop.bat` file again and add the JIMI library to `LOCALCLASSPATH`:

   ```
   set LOCALCLASSPATH=%LOCALCLASSPATH%;%LIBDIR%\JimiProClasses.jar
   ```

7. You should now be able to run FOP from a DOS prompt — execute it without arguments to get a list of command options:

   ```
   > fop.bat
   ```

E. Installing FOP on Linux

Here's how I installed FOP on Fedora Core 1:

1. Download latest FOP distribution from http://xml.apache.org/fop/.

2. Install the FOP distribution:

   ```
   $ su
   # cd /usr/local/lib
   # unzip ~srackham/tmp/fop-0.20.5-bin.zip
   # cp /usr/local/lib/fop-0.20.5/fop.sh /usr/local/bin
   # chmod +x /usr/local/bin/fop.sh
   ```

3. Edit the FOP start script `fop.sh` adding this line to the start of the script:

   ```
   FOP_HOME=/usr/local/lib/fop-0.20.5
   ```

4. Download the JIMI image processing library from http://java.sun.com/products/jimi/.

5. Extract the `JimiProClasses.jar` library from the JIMI distribution and copy to the FOP `lib` directory.

   ```
   # cp ~srackham/tmp/JimiProClasses.jar /usr/local/lib/fop-0.20.5/lib/
   ```

6. You should now be able to run FOP from a DOS prompt — execute it without arguments to get a list of command options:

   ```
   $ fop.sh
   ```

F. Installing Java on Windows

First check that Java is not already installed:

1. Open a DOS *Command Prompt* window.
2. Enter this command:

   ```
   java -version
   ```

You should see something like this:

```
java version "1.4.2_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_01-b06)
Java HotSpot(TM) Client VM (build 1.4.2_01-b06, mixed mode)
```

If you don't Java is not installed and you need to:

1. Download the Java Runtime (JRE) for Windows from http://java.sun.com.
2. Install using the instructions on the download page.

G. Installing Java on Linux

Check Java is not already installed by entering the following command:

```
$ java -version
```

You should see something like this:

```
java version "1.4.2_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_01-b06)
Java HotSpot(TM) Client VM (build 1.4.2_01-b06, mixed mode)
```

If it's not already set you will need to set the `JAVA_HOME` environment variable. For example on Kubuntu Breezy put the following line into `/etc/profile`:

```
export JAVA_HOME=/usr/lib/jvm/java-1.4.2-gcj-4.0-1.4.2.0/
```

H. Using AsciiDoc with non-English Languages

AsciiDoc can process UTF-8 character sets but there are some things you need to be aware of:

- If you are generating output documents using a DocBook toolchain then you should set the *AsciiDoc* `lang` attribute to the appropriate language (it defaults to `en` (English)). This will ensure things like table of contents, revision history, figure and table captions and admonition captions are output in the specified language. For example:

  ```
  $ a2x -a lang=es doc/article.txt
  ```

- If you are outputting html or xhtml directly from `asciidoc(1)` you'll need to set the various `*_caption` attributes to match your target language. The easiest way is to create a language `.conf` file (see the example `lang-es.conf` file that comes with the *AsciiDoc* distribution).

- `asciidoc(1)` automatically loads configuration files named like `lang-<lang>.conf` where `<lang>` is a two letter language code that matches the current *AsciiDoc* `lang` attribute. See also Configuration File Names and Locations.

- Some character sets display double-width characters (for example Japanese). As far as title underlines are concerned they should be treated as single character. If you think this looks untidy so you may prefer to use the single line title format.

I. ASCIIMathML Support

ASCIIMathML [http://www1.chapman.edu/~jipsen/mathml/asciimath.html] is a clever JavaScript written by Peter Jipsen that transforms mathematical formulae written in plain text to standard mathematical notation on an HTML page.

To enable ASCIIMathML support on the `xhtml11` backend include the `-a asciimath` command-line option. Here's what the `asciimath` attribute does:

- Embeds the `ASCIIMathML.js` script in the output document (links it if `-a linkcss` has been specified).

- Escapes ASCIIMathML delimiters.

When entering ASCIIMathML formulas you **must** enclose them inside double-dollar passthroughs (this is necessary because ASCIIMathML characters clash with *AsciiDoc* formatting characters). The double-dollar passthrough has the bonus of also escaping special characters so the output document is valid XHTML. You can see an ASCIIMathML example at http://www.methods.co.nz/asciidoc/asciimath.html, the same example can be found in the *AsciiDoc* distribution `./doc` directory.

- See the ASCIIMathML [http://www1.chapman.edu/~jipsen/mathml/asciimath.html] website for ASCIIMathML documentation and the latest version.

- If you use Mozilla you need to install the required math fonts [http://www.mozilla.org/projects/mathml/fonts/].

- If you use Microsoft Internet Explorer 6 you need to install MathPlayer [http://www.dessci.com/en/products/mathplayer/].

J. Vim Syntax Highlighter

The *AsciiDoc* `./vim/` distribution directory contains Vim syntax highlighter and filetype detection scripts for *AsciiDoc*. Syntax highlighting makes it much easier to spot *AsciiDoc* syntax errors.

If Vim is installed on your system the *AsciiDoc* installer (`install.sh`) will automatically install the vim scripts in the Vim global configuration directory (`/etc/vim`).

You can also turn on syntax highlighting by adding the following line to the end of you *AsciiDoc* source files:

```
// vim: set syntax=asciidoc:
```

Dag Wieers has implemented an alternative Vim syntax file for *AsciiDoc* which can be found here http://svn.rpmforge.net/svn/trunk/tools/asciidoc-vim/.

Emacs users: The *Nix Power Tools project [http://xpt.sourceforge.net/] has released an *AsciiDoc* syntax highlighter for emacs [http://xpt.sourceforge.net/tools/doc-mode/].

Limitations

The current implementation does a reasonable job but on occasions gets things wrong. This list of limitations also discusses how to work around the problems:

- Indented lists with preceding blank lines are sometimes mistaken for literal (indented) paragraphs. You can work around this by deleting the preceding blank line, or inserting a space in the preceding blank lines, or putting a list continuation character (+) in the preceding blank line.
- Nested text formatting is highlighted according to the outer format.
- Text formatting is not highlighted inside titles or attribute lists.
- Most escaped inline elements will be highlighted.
- Unterminated quotes are highlighted, for example `'tis` would be seen as the start of emphasized text. In this case work-around would be to comment out `asciidocEmphasized2` and use the (`asciidocEmphasized`) underscored for emphasis. As a damage control measure quoted patterns always terminate at a blank line. This problem is usually ameliorated by the fact that characters such as ~, +, ^ and _ will normally occur inside monospaced quotes (unless they are used for quoting), for example `~/projects`.
- If a closing block delimiter is not preceded by a blank line it is sometimes mistaken for a title underline. A workaround is to insert a blank line before the closing delimiter.
- If a list block delimiter is mistaken for a title underline precede it with a blank line.
- Tables are terminated by a blank line — use a space character on blank lines within your table.

- Lines within a paragraph beginning with a period will be highlighted as block titles. For example:

  ```
  .chm file.
  ```

  To work around this restriction move the last word of the previous line to the start of the current (although words starting with a period should probably be quoted monospace which would also get around the problem).

Sometimes incorrect highlighting is caused by preceding lines that appear blank but contain white space characters.