Part I. Quickbook 1.6

Table of Contents

Introduction

> **“Why program by hand in five days what you can spend five years of your life automating?”**
>
> -- Terrence Parr, author ANTLR/PCCTS

Well, QuickBook started as a weekend hack. It was originally intended to be a sample application using Spirit. What is it? What you are viewing now, this documentation, is autogenerated by QuickBook. These files were generated from one master:

quickbook.qbk

Originally named QuickDoc, this funky tool that never dies, evolved into a funkier tool thanks to Eric Niebler who resurrected the project making it generate BoostBook instead of HTML. The BoostBook documentation format is an extension of DocBook, an SGML or XML based format for describing documentation.

Tip

You don't need to know anything about BoostBook or DocBook to use QuickBook. A basic understanding of DocBook might help, but shouldn't be necessary. For really advanced stuff you will need to know DocBook, but you can ignore it at first, and maybe continue to do so.

QuickBook is a WikiWiki style documentation tool geared towards C++ documentation using simple rules and markup for simple formatting tasks. QuickBook extends the WikiWiki concept. Like the WikiWiki, QuickBook documents are simple text files. A single QuickBook document can generate a fully linked set of nice HTML and PostScript/PDF documents complete with images and syntax- colorized source code.

Features include:

- generate BoostBook xml, to generate HTML, PostScript and PDF
- simple markup to link to Doxygen-generated entities
- macro system for simple text substitution
- simple markup for italics, bold, preformatted, blurbs, code samples, tables, URLs, anchors, images, etc.
- automatic syntax coloring of code samples
- CSS support

Change Log

Version 1.1 - Boost 1.33.0

- First version to be included in boost.

Version 1.3 - Boost 1.34.0 to 1.34.1

- Quickbook file inclusion [include].
- Better xml output (pretty layout). Check out the generated XML.
- Regression testing facility: to make sure your document will always be compatible (full backward compatibility) regardless of changes to QuickBook.
- Code cleanup and refactoring.
- Allow phrase markup in the doc-info.
- Preformatted code blocks via ``code`` (double ticks) allows code in tables and lists, for example.
- Quickbook versioning; allows full backward compatibility. You have to add [quickbook 1.3] to the doc-info header to enable the new features. Without this, QuickBook will assume that the document is a pre-1.3 document.
- Better (intuitive) paragraph termination. Some markups may terminate a paragraph. Example:

```
[section x]
blah...
[endsect]
```

- Fully qualified section and headers. Subsection names are concatenated to the ID to avoid clashing. Example: `doc_name.sect_name.sub_sect_name.sub_sub_sect_name`
- Better   and whitespace handling in code snippets.
- [xinclude] fixes up the relative path to the target XML file when input_directory is not the same as the output_directory.
- Allow untitled tables.
- Allow phrase markups in section titles.
- Allow escaping back to QuickBook from code, code blocks and inline code.
- Footnotes, with the [footnote This is the footnote] syntax.
- Post-processor bug fix for escaped XML code that it does not recognize.
- Replaceable, with the [~replacement] syntax.

Version 1.4 - Boost 1.35.0 to 1.40.0

- Generic Headers
- Code changes to allow full recursion (i.e. Collectors and push/pop functions)
- Various code cleanup/maintenance

- Templates!
- [conceptref] for referencing BoostBook <concept> entities.
- Allow escape of spaces. The escaped space is removed from the output. Syntax: `\ `.
- Nested comments are now allowed.
- Quickbook blocks can nest inside comments.
- Import facility.
- Callouts on imported code
- Simple markups can now span a whole block.
- Blurbs, Admonitions and table cells (see Tables) may now contain paragraphs.
- `\n` and `[br]` are now deprecated.
- Conditional Generation. Ala C++ #ifdef.
- Searching of included and imported files in an extensible search path with `--include-path` (`-I`) option.

Version 1.5 - Boost 1.41.0 to 1.42.0

- Support multiple copyright entrys in document info.
- Improved SVG support.
- [globalref] for referencing BoostBook <global> entities.
- Fail on error.
- Fix crash for templates with too many arguments or trailing space.
- Improved handling of unexpected characters in code blocks.
- Improved handling of unmatched escape in code blocks.
- Support for python snippets.
- `teletype` source mode.
- Use static scoping in templates, should be a lot more intuitive.
- Accept a space between `section:` and the section id.
- Support table ids.

Version 1.5.1 - Boost 1.43.0

- Improve the post processor's list of block elements. `table`, `entry` and `varlistentry` are treated as blocks. `replaceable` is treated as an inline element.
- Check that `[section]` and `[endsect]` tags are balanced in templates.
- Add unicode escape characters, eg. `\u03B1` for α.
- Support UTF-8 files with a unicode byte order mark.

- Disallow `[` in simple markup. Fixes some errors with mismatched punctuation.
- Add command line flag to define macros at the command line, e.g. `quickbook "-D__italic_foo__=/foo/"`.

Version 1.5.2 - Boost 1.44.0

- Use the cygwin 1.7 API for better path handling.
- Improved boostbook generation:
  - XML encode the documentation info correctly.
  - Avoid generating empty paragraphs.
  - No longer wraps block templates in paragraphs.
  - Warns if you use invalid doc_info members for docbook document types.
  - Fixes some other causes of invalid boostbook, although it still generates invalid boostbook in places.
- Improved grammar:
  - Supports multiple categories in library doc_info.
  - No longer requires commas between authors in docinfo.
  - Allows empty document bodies.
  - A line containing only a comment is no longer interpreted as a paragraph break.
  - If a line starts with a comment, interpret it as a paragraph even if it's followed by whitespace or a list character.
  - Doesn't treat several consecutive blank lines as multiple paragraph breaks.
- Fixes duplicate image attribute detection.
- Fixes using code snippets more than once.
- Early work on quickbook 1.6, available using the `[quickbook 1.6]` version switch, but liable to change in future versions.
  - When automatically generating ids for headers, use the quickbook source, rather than the generated docbook.
  - Fix id generation in included files. It wasn't correctly using the main document's documentation id.
  - Correctly restore the quickbook version switch after including a file with a different version.

Version 1.5.3 - Boost 1.45.0

- Fix command line flag for defining macros.
- Fix a couple of issues with the code block parser:
  - A comment with no indentation will now end a code block.
  - Code blocks no longer have to be followed by a blank line.
- Improved tracking of file position in templates and imported code blocks.
- Better generated markup for callout lists.
- In docbook, variable list entries can only have one `listitem`, so if an entry has multiple values, merge them into one `listitem`.

- Support nested code snippets.
- Support nested blocks in document info comments.
- Revert xml escaping document info, it broke some documentation files (now a 1.6 feature).
- Further work on quickbook 1.6, still not stable.
  - Allow heading to have ids, using the syntax: `[heading:id title]`.
  - XML escape documentation fields, with escapes to allow encoding unicode in ASCII.

Version 1.5.4 - Boost 1.46.1

Boost 1.46.0:

- Add support for `lang` attribute in documentation info.
- Improved anchor implementation. Especially for using an anchor before a section or heading.
- Fixed some more issues where lines containing comments were treated as blank lines.
- Allow import, include and xinclude in conditional phrases. Will allow more block elements in a future version.
- Rearrange the structure of the grammar.
- Use filesystem 3. Remove cygwin 1.5 support.

Boost 1.46.1:

- Work around optimization bug in g++ 4.4 on 64 bit linux.

Version 1.5.5 - Boost 1.47

- Tweak anchor placement for titles.
- Hard code the quickbook path into the quickbook testing tools. This means that they can be used from multiple locations.
- Generate an id for boostbook `bridgehead` elements. This results in more consistent html, since docbook generates a random id if they don't have one.
- Improved unicode support on windows. Unicode can now be used from the command line, and unicode filenames are supported. Unicode output is a bit weak.
- Check for windows paths, and warn about them.
- Fix relative path detection on windows.
- Reverse deprecation of `[br]`, printing a single warning about generating invalid boostbook.
- Fix handling empty category attributes.
- Store data from the parser in a dynamic data structure. This simplifies the implementation and makes it easier to parse more complicated data structures.
- Improved error messages for unknown doc info attributes.
- Richer copyright syntax. Now understands: `[copyright 2001-2006, 2010 One person, 2008 Another person]`.
- Fix delimeter checking for simple markup.

- Allow more block elements to be nested.
- Go back to using invalid markup for lists. It generates better html.
- Better anchor placement for lists.
- Pass-thru comments in code snippets.
- Use relative paths for `__FILENAME__` macro.
- Rewrite xinclude path generator so that it doesn't use deprecated filesystem functions.
- Allow quickbook escapes inside comments in syntax highlighted code.
- Quickbook 1.6:
  - Scope source mode changes to the file they're made in.
  - Explicit markup for lists. e.g. `[ordered_list [item1][item2]]` or `[itemized_list [item1][item2]]`.

Version 1.5.6 - Boost 1.48

- Xml encode escaped punctuation (eg. \< is correctly encodes to <).
- Rename duplicate generated ids.
- Close open sections at end of document (still warns about them).
- New anchor markup for headers, will hopefully generate better pdfs.
- Remove some whitespace around code from post processed output.

Version 1.5.7 - Boost 1.49

- Several internal changes.
- Some improved error messages.
- Better handling of block templates expanded in a phrase context.
- Avoids empty simple markup (i.e. // is not treated as an italic empty space.
- Better anchor markup for headers, which should be better for printing - suggested by John Maddock.
- Further improvements to the id generator.
- If sections are left unopened at the end of a document, then close them in the generated markup.
- Try to handle whitespace better at the beginning and end of code blocks.
- Handle lists that come immediately after an anchor.
- Make horizontal rules followed by multi-line comments a little more sensible.
- Better support for empty ids and titles in docinfo.
- Fix some minor regressions in SVG handling.
- Better handling of invalid command line macros.
- When auto-building quickbook, build the release version.

- Lots of changes for 1.6:
  - Scope templates in included files.
  - Support import of templates and macros.
  - Including top level quickbook blocks from source files.
  - Use doc info blocks in included quickbook files.
  - Better handling of macros with the same name.
  - `block` element.
  - Better handling of significant punctuation (e.g. escapes, square brackets).
  - Support escapes in links, anchors, images, includes etc.
  - Improved table title syntax.
  - Paragraphs nested in lists.
  - New docinfo attributes:
    - `compatibility-mode` to make it possible to upgrade documents without breaking ids.
    - `xmlbase` for escaped `xi:includes`.
  - Allow some docinfo attributes to be used before, or without, a doc info block (`quickbook`, `compatibility-mode`, `source-mode`).
  - Only add explicit alt text to images.
  - Don't put 'inline' code blocks inside paragraphs.

Version 1.5.8 - Boost 1.50

- Write dependencies to a file, using `--output-deps` (#6691).
- Fix handling of section tags in lists.
- Fix indented code blocks in lists.
- Fix handling UTF-8 code points in the syntax highlighter. Was treating each individual byte as a character. Still doesn't deal with combining code points.
- Internal changes:
  - A lot of restructuring.
  - Stop using 'v3' filesystem paths and namespaces, it's now the default version.
  - Remove awkward intrusive reference counting implementation, avoids a gcc internal compiler error (#6794), but is also a cleaner implementation.
- 1.6 changes:
  - Better handling of brackets in link values.
  - Improved handling of escaped characters in include paths.
- Starting to develop 1.7:

- Source mode for single entities.
- Callouts in code blocks.
- Escaped docbook in docinfo blocks.
- Starting to implement calling templates from link values.

Version 1.5.9 - Boost 1.54

- When code blocks are indented using a mixture of tabs and spaces, convert indentation to spaces.
- In the C++ systax highlighter, fix syntax highlighting for #, so that it's used for preprocessor statements at the start of a line, and as a 'special' character elsewhere (#8510, #8511).
- Add C++11 keywords to syntax highlighter (#8541).
- Hidden options for formatting of `--output-deps`. Not really for public use
- yet.
- 1.6 changes:
  - Better template argument parsing, so that it understands things like escaped markup.
  - Support for using macros in the doc info block.
- Internal changes:
  - Convert to use `boost::string_ref`.
  - Clean up the source map implementation (used to get the correct location for error messages in things like templates and snippets).

Version 1.6.0 - Boost 1.55

- Remove nested blocks in lists from 1.6, move to 1.7. (Can still nest block elements in lists though).
- Don't break out of lists after a nested block element.
- Check for errors when writing dependency files.
- Improved markup for lists.
- Make escaping templates with a punctuation identifier illegal. Escaping templates with an alphanumeric identifier is still fine.
- Fix detection of code blocks at the start of a file.
- XML encode the contents of the `change_log.qbk` macro.
- 1.7 changes:
  - Make it an error to use an element in the wrong context.
  - Error if the body of a phrase element doesn't parse.
  - List markup in nested blocks.
  - Allow block elements in phrase templates.
  - Make it an error to put a paragraph break (i.e. a blank line) in a phrase template.

- Internal changes:
  - Clean up the id manager implementation.

Version 1.6.1

- Better URI encoding of links.
- Extra validation of attribute values.
- 1.7 changes:
  - Improved source mode tagging:
    - Works for lists and paragraphs.
    - If the source mode is changed inside a tagged element, that change will now persist after the element.
    - Tagged sections will now use the source mode for the whole section.
  - Template calls from anchor, role and include elements.
  - Stricter handling of templates called in attribute values.
  - Glob support.

Syntax Summary

A QuickBook document is composed of one or more blocks. An example of a block is the paragraph or a C++ code snippet. Some blocks have special mark-ups. Blocks, except code snippets which have their own grammar (C++ or Python), are composed of one or more phrases. A phrase can be a simple contiguous run of characters. Phrases can have special mark-ups. Marked up phrases can recursively contain other phrases, but cannot contain blocks. A terminal is a self contained block-level or phrase-level element that does not nest anything.

Blocks, in general, are delimited by two end-of-lines (the block terminator). Phrases in each block cannot contain a block terminator. This way, syntax errors such as un-matched closing brackets do not go haywire and corrupt anything past a single block.

Comments

Can be placed anywhere.

```
[/ comment (no output generated) ]
```

```
[/ comments can be nested [/ some more here] ]
```

```
[/ Quickbook blocks can nest inside comments. [*Comment this out too!] ]
```

Document Structure

Document Info

Every document must begin with a Document Info section, which looks something like this:

```
[article The Document Title
    [quickbook 1.5]
    [version 1.0]
    [id the_document_name]
    [copyright 2000 2002 2003 Joe Blow, Jane Doe]
    [authors [Blow, Joe] [Doe, Jane]]
    [license The document's license]
    [source-mode c++]
]
```

`article` is the document type. There are several possible document types, most of these are based on docbook document elements. These are fully described in DocBook: The Definitive Guide:

- book
- article
- chapter
- part
- appendix
- preface
- qandadiv
- qandaset
- reference
- set

Boostbook also adds another document type `library` for documenting software libraries.

So the documentation for the 'foo' library might start:

```
[library Foo
    [quickbook 1.5]
    [id foo]
    [version 1.0]
]
```

Document Info Attributes

The document info block has a few different types of attributes. They are all optional.

Quickbook specific meta data

```
[quickbook 1.6]
```

The `quickbook` attribute declares the version of quickbook the document is written for. In its absence, version 1.1 is assumed. It's recommended that you use `[quickbook 1.6]` which is the version described here.

Note

The quickbook version also makes some changes to the markup that's generated. Most notably, the ids that are automatically for headers and sections are different in later versions. To minimise disruption, you can use the `compatibility-mode` attribute to generate similar markup to the old version:

```
[article Article that was original
         written in quickbook 1.3
[quickbook 1.6]
[compatibility-mode 1.3]
]
```

This feature shouldn't be used for new documents, just for porting old documents to the new version.

Both the `quickbook` and `compatibility-mode` tags can be used at the start of the file, before the document info block, and also in files that don't have a document info block.

```
[source-mode teletype]
```

The `source-mode` attribute sets the initial Source Mode. If it is omitted, the default value of `c++` will be used.

Boostbook/Docbook root element attributes

```
[id foo]
```

`id` specifies the id of the document element. If it isn't specified the id is automatically generated from the title. This id is also used to generate the nested ids.

```
[lang en]
```

`lang` specifies the document language. This is used by docbook to localize the documentation. Note that Boostbook doesn't have any localization support so if you use it to generate the reference documentation it will be in English regardless.

It should be a language code drawn from ISO 639 (perhaps extended with a country code drawn from ISO 3166, as en-US).

```
[dirname foo]
```

`dirname` is used to specify the directory name of the library in the repository. This is a boostbook extension so it's only valid for `library` documentation blocks. It's used for some boostbook functionality, but for pure quickbook documentation has no practical effect.

Docbook Metadata

`version`, `copyright`, `authors`, `license`, `last-revision` and `bibliod` are optional information.

Boostbook Metadata

`purpose` and `category` are boostbook attributes which are only valid for `library` documents. If you use them for other document types, quickbook will warn about them, but still use them, generating invalid markup, that's just ignored by the style sheets.

Nesting quickbook documents

Docinfo blocks can only appear at the beginning of a quickbook file, so to create a more complicated document you need to use several quickbook files and use the include tag to nest them. For example, say you wish to create a book with an introduction and a chapter, you first create a file for the book:

```
[book Simple example
[quickbook 1.6]
]

[include introduction.qbk]
[include chapter.qbk]
```

Note

Structuring a document like this was introduced in quickbook 1.6, so the `[quickbook 1.6]` docinfo field is required.

The appropriate document type for an introduction is `preface`, so the contents of `introduction.qbk` should be something like:

```
[preface Introduction
[quickbook 1.6]
]

Write the introduction to the book here....
```

And `chapter.qbk`:

```
[chapter A chapter
[quickbook 1.6]
]

Chapter contents....
```

Sections

Quickbook documents are structured using 'sections'. These are used to generate the table of contents, and, when generating html, to split the document into pages. This is optional but a good idea for all but the simplest of documents.

A sectioned document might look like:

```
[book Title
    [quickbook 1.5]
]

[section First Section]

[/...]

[endsect]

[section Second Section]

[/...]

[endsect]
```

Sections start with the `section` tag, and end with the `[endsect]` tag. (`[/...]` is a comment, described later).

Sections can be given an optional id:

```
[#quickbook.ref.id]
[section:id The Section Title]
```

`id` will be the filename of the generated section. If it is not present, "The Section Title" will be normalized and become the id. Valid characters are `a-z`, `A-Z`, `0-9` and _. All non-valid characters are converted to underscore and all upper-case are converted to lower case. Thus: "The Section Title" will be normalized to "the_section_title".

Sections can nest, and that results in a hierarchy in the table of contents.

Phrase Level Elements

Font Styles

```
['italic], [*bold], [_underline], [^teletype], [-strikethrough]
```

will generate:

italic, **bold**, underline, `teletype`, ~~strikethrough~~

Like all non-terminal phrase level elements, this can of course be nested:

```
[*['bold-italic]]
```

will generate:

bold-italic

Replaceable

When you want content that may or must be replaced by the user, use the syntax:

```
[~replacement]
```

This will generate:

replacement

Quotations

```
["A question that sometimes drives me hazy: am I or are the others crazy?]--Einstein
```

will generate:

“A question that sometimes drives me hazy: am I or are the others crazy?”--Einstein

Note the proper left and right quote marks. Also, while you can simply use ordinary quote marks like "quoted", our quotation, above, will generate correct DocBook quotations (e.g. <quote>quoted</quote>).

Like all phrase elements, quotations may be nested. Example:

```
["Here's the rule for bargains: ["Do other men, for they would do you.] That's
the true business precept.]
```

will generate:

“Here's the rule for bargains: ‘Do other men, for they would do you.’ That's the true business precept.”

Simple formatting

Simple markup for formatting text, common in many applications, is now supported:

```
/italic/, *bold*, _underline_, =teletype=
```

will generate:

italic, **bold**, underline, `teletype`

Unlike QuickBook's standard formatting scheme, the rules for simpler alternatives are much stricter[1].

- Simple markups cannot nest. You can combine a simple markup with a nestable markup.
- Simple markups cannot contain any other form of quickbook markup.
- A non-space character must follow the leading markup
- A non-space character must precede the trailing markup
- A space or a punctuation must follow the trailing markup
- If the matching markup cannot be found within a block, the formatting will not be applied. This is to ensure that un-matched formatting markups, which can be a common mistake, does not corrupt anything past a single block. We do not want the rest of the document to be rendered bold just because we forgot a trailing '*'. A single block is terminated by two end of lines or the close bracket: ']'.
- A line starting with the star will be interpreted as an unordered list. See Unordered lists.

[1] Thanks to David Barrett, author of Qwiki, for sharing these samples and teaching me these obscure formatting rules. I wasn't sure at all if Spirit, being more or less a formal EBNF parser, can handle the context sensitivity and ambiguity.

Table 1. More Formatting Samples

Markup	Result
`*Bold*`	**Bold**
`*Is bold*`	**Is bold**
`* Not bold* *Not bold * * Not bold *`	* Not bold* *Not bold * * Not bold *
`This*Isn't*Bold (no bold)`	This*Isn't*Bold (no bold)
`(*Bold Inside*) (parenthesis not bold)`	(**Bold Inside**) (parenthesis not bold)
`*(Bold Outside)* (parenthesis bold)`	**(Bold Outside)** (parenthesis bold)
`3*4*5 = 60 (no bold)`	3*4*5 = 60 (no bold)
`3 * 4 * 5 = 60 (no bold)`	3 * 4 * 5 = 60 (no bold)
`3 *4* 5 = 60 (4 is bold)`	3 **4** 5 = 60 (4 is bold)
`*This is bold* this is not *but this is*`	**This is bold** this is not **but this is**
`*This is bold*.`	**This is bold**.
`*B*. (bold B)`	**B**. (bold B)
`['*Bold-Italic*]`	***Bold-Italic***
`*side-by*/-side/`	**side-by**-*side*

As mentioned, simple markups cannot go past a single block. The text from "have" to "full" in the following paragraph will be rendered as bold:

```
Baa baa black sheep, *have you any wool?
Yes sir, yes sir, three bags full!*
One for the master, one for the dame,
And one for the little boy who lives down the lane.
```

Baa baa black sheep, **have you any wool? Yes sir, yes sir, three bags full!** One for the master, one for the dame, And one for the little boy who lives down the lane.

But in the following paragraph, bold is not applied:

```
Baa baa black sheep, *have you any wool?
Yes sir, yes sir, three bags full!
One for the master, one for the dame,
And one for the little boy who lives down the lane.
```

Baa baa black sheep, *have you any wool? Yes sir, yes sir, three bags full! One for the master, one for the dame, And one for the little boy who lives down the lane.

Role

This generates a docbook phrase with a `role` attribute, which can be used to classify the phrase. This can be used to mark text for a use that isn't covered elsewhere. The docbook `role` will generate a html class, which can be used to style text. And the xsl stylesheets can be customized to treat certain roles specially when generating pdfs.

The boostbook css stylesheets, and xsl stylesheets contain support for a limited number of colours that can be used with `role`. For example if you write:

```
[role red Text content]
```

You'll get red text if you're using the boostbook css (for html) or the boostbook xsl for generating pdfs.

The full list of colours that will be available is:

- red
- green
- lime
- blue
- navy
- yellow
- magenta
- indigo
- cyan
- purple
- gold
- silver
- gray

Inline code

Inlining code in paragraphs is quite common when writing C++ documentation. We provide a very simple markup for this. For example, this:

```
This text has inlined code `int main() { return 0; }` in it.
```

will generate:

This text has inlined code `int main() { return 0; }` in it. The code will be syntax highlighted.

Note

We simply enclose the code with the tick: "`", not the single quote: "'". Note too that `` `some code` `` is preferred over `[^some code]`.

Code blocks

Preformatted code simply starts with a space or a tab (See Code). However, such a simple syntax cannot be used as phrase elements in lists (See Ordered lists and Unordered lists), tables (See Tables), etc. Inline code (see above) can. The problem is, inline code does not allow formatting with newlines, spaces, and tabs. These are lost.

We provide a phrase level markup that is a mix between the two. By using the double-tick or triple-tick, instead of the single-tick, we are telling QuickBook to use preformatted blocks of code. Example:

```
``
    #include <iostream>

    int main()
    {
        std::cout << "Hello, World!" << std::endl;
        return 0;
    }
``
```

or:

```
```
    #include <iostream>

    int main()
    {
        std::cout << "Hello, World!" << std::endl;
        return 0;
    }
```
```

will generate:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Source Mode

If a document contains more than one type of source code then the source mode may be changed dynamically as the document is processed. All QuickBook documents are initially in C++ mode by default, though an alternative initial value may be set in the Document section.

To change the source mode, use the `[source-mode]` markup, where `source-mode` is one of the supported modes. For example, this:

```
Python's [python] `import` is rather like C++'s [c++] `#include`. A
C++ comment `// looks like this` whereas a Python comment [python]
`# looks like this`.
```

will generate:

Python's `import` is rather like C++'s `#include`. A C++ comment `// looks like this` whereas a Python comment `#looks like this`.

Table 2. Supported Source Modes

Mode	Source Mode Markup
C++	`[c++]`
Python	`[python]`
Plain Text	`[teletype]`

Note

The source mode strings are lowercase.

line-break

```
[br]
```

Warning

`[br]` generates invalid docbook. It seems to mostly work okay but there might be problems, especially when using an alternative docbook processor.

Anchors

```
[#named_anchor]
```

A named anchor is a hook that can be referenced by a link elsewhere in the document. You can then reference an anchor with `[link named_anchor Some link text]`. See Anchor links, Section and Heading.

These anchors are global and can be accessed from anywhere in the quickbook documentation. Be careful to avoid clashes with anchors in other sections.

Links

```
[@http://www.boost.org this is [*boost's] website....]
```

will generate:

this is **boost's** website....

URL links where the link text is the link itself is common. Example:

```
see http://spirit.sourceforge.net/
```

so, when the text is absent in a link markup, the URL is assumed. Example:

```
see [@http://spirit.sourceforge.net/]
```

will generate:

see http://spirit.sourceforge.net/

Boostbook also support a custom url schema for linking to files within the boost distribution:

```
[@boost:/libs/spirit/index.html the Boost.Spirit documentation]
```

will generate: the Boost.Spirit documentation

Note that this is only available when using BoostBook, and only for links - it can't be used for images.

Anchor links

You can link within a document using:

```
[link document_id.section_id.normalized_header_text The link text]
```

See sections Section and Heading for more info.

refentry links

In addition, you can link internally to an XML refentry like:

```
[link xml.refentry The link text]
```

This gets converted into `<link linkend="xml.refentry">The link text</link>`.

Like URLs, the link text is optional. If this is not present, the link text will automatically be the refentry. Example:

```
[link xml.refentry]
```

This gets converted into `<link linkend="xml.refentry">xml.refentry</link>`.

Code Links

If you want to link to a function, class, member, enum, concept, global, or header in the reference section, you can use:

```
[funcref fully::qualified::function_name The link text]
[classref fully::qualified::class_name The link text]
[memberref fully::qualified::member_name The link text]
[enumref fully::qualified::enum_name The link text]
[macroref MACRO_NAME The link text]
[conceptref ConceptName The link text]
[headerref path/to/header.hpp The link text]
[globalref fully::qualified::global The link text]
```

Again, the link text is optional. If this is not present, the link text will automatically be the function, class, member, enum, macro, concept, global, or header name. Example:

```
[classref boost::bar::baz]
```

would have "boost::bar::baz" as the link text.

Escape

The escape mark-up is used when we don't want to do any processing.

```
'''
escape (no processing/formatting)
'''
```

Escaping allows us to pass XML markup to BoostBook or DocBook. For example:

```
'''
<emphasis role="bold">This is direct XML markup</emphasis>
'''
```

This is direct XML markup

Important

Be careful when using the escape. The text must conform to BoostBook/DocBook syntax.

Single char escape

The backslash may be used to escape a single punctuation character. The punctuation immediately after the backslash is passed without any processing. This is useful when we need to escape QuickBook punctuations such as `[` and `]`. For example, how do you escape the triple quote? Simple: `\'\'\'`

`\n` has a special meaning. It is used to generate line breaks.

Warning

`\n` is now deprecated, use `[br]` instead. Although, use it sparingly as it can generated invalid docbook

The escaped space: `\ ` also has a special meaning. The escaped space is removed from the output.

Unicode escape

You can enter any 16-bit unicode character by using `\u` followed by its 4 digit hexadecimal code, or a 32-bit character by using `\U` followed by an 8 digit hexadecimal code. eg.

```
\u03B1 + \u03B2
```

will generate:

α + β

Images

```
[$image.jpg]
```

From version 1.5, you can also use DocBook imagedata attributes:

```
[$image.jpg [width 200in] [height 200in]]
```

Footnotes

As of version 1.3, QuickBook supports footnotes. Just put the text of the footnote in a `[footnote]` block, and the text will be put at the bottom of the current page. For example, this:

```
[footnote A sample footnote]
```

will generate this[2].

[2] A sample footnote

Macro Expansion

```
__a_macro_identifier__
```

See Macros for details.

Template Expansion

```
[a_template_identifier]
```

See Templates for details.

Conditional Generation

Like C++ `#ifdef`, you can generate phrases depending on the presence of a macro. Example:

```
[? __to_be__ To be or not to be]
```

Here, the phrase "To be or not to be" will only be generated if the macro symbol `__to_be__` has been previously defined. The phrase above will not do anything since we haven't defined `__to_be__`. Now, let's define the symbol:

```
[def __to_be__]
```

And try again:

To be or not to be

Yes!

Block Level Elements

xinclude

You can include another XML file with:

```
[xinclude file.xml]
```

This is useful when file.xml has been generated by Doxygen and contains your reference section.

`xinclude` paths are normally used unchanged in the generated documentation, which will not work if you wish them to be relative to the current quickbook file. Quickbook can add a `xml:base` attribute to the boostbook documentation to specify where `xinclude` files should be found. For example, if you wish them to be relative to the current quickbook file:

```
[article Article with xincludes
[quickbook 1.6]
[xmlbase .]
]

[xinclude file.xml]
```

Now the xinclude should work if `file.xml` is in the same directory as the quickbook file. Although it might not work if you distribute the generated files (as their relative directories can change).

Say the article is generated in a sub-directory, by running something like:

```
quickbook article.qbk --output-file=output/article.xml
```

This will generate a boostbook root tag:

```
<article id="article_with_xincludes"
    last-revision="$Date: 2013/08/20 08:26:48 $"
    xml:base=".."
    xmlns:xi="http://www.w3.org/2001/XInclude">
```

Because `xml:base` is set to `..`, the xml processor will know to look in the parent directory to find `file.xml`, which it comes across the `xi:include` tag.

Paragraphs

Paragraphs start left-flushed and are terminated by two or more newlines. No markup is needed for paragraphs. QuickBook automatically detects paragraphs from the context. Block markups [section, endsect, h1, h2, h3, h4, h5, h6, blurb, (block-quote) ':', pre, def, table and include] may also terminate a paragraph. This is a new paragraph...

Lists

Ordered lists

```
# One
# Two
# Three
```

will generate:

1. One
2. Two
3. Three

List Hierarchies

List hierarchies are supported. Example:

```
# One
# Two
# Three
    # Three.a
    # Three.b
    # Three.c
# Four
    # Four.a
        # Four.a.i
        # Four.a.ii
# Five
```

will generate:

1. One
2. Two
3. Three
    a. Three.a
    b. Three.b
    c. Three.c
4. Fourth
    a. Four.a
        i. Four.a.i
        ii. Four.a.ii
5. Five

Long List Lines

Long lines will be wrapped appropriately. Example:

```
# A short item.
# A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
  A very long item. A very long item. A very long item.
# A short item.
```

1. A short item.
2. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item. A very long item.
3. A short item.

Unordered lists

```
* First
* Second
* Third
```

will generate:

- First
- Second
- Third

Mixed lists

Mixed lists (ordered and unordered) are supported. Example:

```
# One
# Two
# Three
    * Three.a
    * Three.b
    * Three.c
# Four
```

will generate:

1. One
2. Two
3. Three
    - Three.a
    - Three.b

- Three.c

4. Four

And...

```
# 1
    * 1.a
        # 1.a.1
        # 1.a.2
    * 1.b
# 2
    * 2.a
    * 2.b
        # 2.b.1
        # 2.b.2
            * 2.b.2.a
            * 2.b.2.b
```

will generate:

1. 1
   - 1.a
     a. 1.a.1
     b. 1.a.2
   - 1.b
2. 2
   - 2.a
   - 2.b
     a. 2.b.1
     b. 2.b.2
        - 2.b.2.a
        - 2.b.2.b

Explicit list tags

Sometimes the wiki-style list markup can be tricky to use, especially if you wish to include more complicated markup with the list. So in quickbook 1.6, an alternative way to mark up lists introduced:

```
[ordered_list [item1][item2]]
```

is equivalent to:

```
# item1
# item2
```

And:

```
[itemized_list [item1][item2]]
```

is equivalent to:

```
* item1
* item2
```

Code

Preformatted code starts with a space or a tab. The code will be syntax highlighted according to the current Source Mode:

```
#include <iostream>

int main()
{
    // Sample code
    std::cout << "Hello, World\n";
    return 0;
}
```

```
import cgi

def cookForHtml(text):
    '''"Cooks" the input text for HTML.'''

    return cgi.escape(text)
```

Macros that are already defined are expanded in source code. Example:

```
[def __array__ [@http://www.boost.org/doc/html/array/reference.html array]]
[def __boost__ [@http://www.boost.org/libs/libraries.htm boost]]

    using __boost__::__array__;
```

Generates:

```
using boost::array;
```

Escaping Back To QuickBook

Inside code, code blocks and inline code, QuickBook does not allow any markup to avoid conflicts with the target syntax (e.g. c++). In case you need to switch back to QuickBook markup inside code, you can do so using a language specific *escape-back* delimiter. In C++ and Python, the delimiter is the double tick (back-quote): "``" and "``". Example:

```
void foo()
{
}
```

Will generate:

```
void foo()
{
}
```

When escaping from code to QuickBook, only phrase level markups are allowed. Block level markups like lists, tables etc. are not allowed.

Preformatted

Sometimes, you don't want some preformatted text to be parsed as source code. In such cases, use the `[pre ... ]` markup block.

```
[pre

    Some *preformatted* text                    Some *preformatted* text

        Some *preformatted* text            Some *preformatted* text

            Some *preformatted* text    Some *preformatted* text

]
```

Spaces, tabs and newlines are rendered as-is. Unlike all quickbook block level markup, pre (and Code) are the only ones that allow multiple newlines. The markup above will generate:

```

Some preformatted text                    Some preformatted text

    Some preformatted text            Some preformatted text

        Some preformatted text    Some preformatted text
```

Notice that unlike Code, phrase markup such as font style is still permitted inside `pre` blocks.

Blockquote

```
[:sometext...]
```

Indents the paragraph. This applies to one paragraph only.

Admonitions

```
[note This is a note]
[tip This is a tip]
[important This is important]
[caution This is a caution]
[warning This is a warning]
```

generates DocBook admonitions:

Note

This is a note

Tip

This is a tip

Important

This is important

Caution

This is a caution

Warning

This is a warning

These are the only admonitions supported by DocBook. So, for example `[information This is some information]` is unlikely to produce the desired effect.

Headings

```
[h1 Heading 1]
[h2 Heading 2]
[h3 Heading 3]
[h4 Heading 4]
[h5 Heading 5]
[h6 Heading 6]
```

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

You can specify an id for a heading:

```
[h1:heading_id A heading to link to]
```

To link to it, you'll need to include the enclosing section's id:

```
[link document_id.section_id.heading_id The link text]
```

Although you can preceed a heading by an anchor if you wish to use a location independent link.

If a heading doesn't have an id, one will be automatically generated with a normalized name with `name="document_id.section_id.normalized_header_text"` (i.e. valid characters are `a-z`, `A-Z`, `0-9` and _. All non-valid characters are converted to underscore and all upper-case are converted to lower-case. For example: Heading 1 in section Section 2 will be normalized to `section_2.heading_1`). You can use:

```
[link document_id.section_id.normalized_header_text The link text]
```

to link to them. See Anchor links and Section for more info.

Note

Specifying heading ids is a quickbook 1.6 feature, earlier versions don't support them.

Generic Heading

In cases when you don't want to care about the heading level (1 to 6), you can use the *Generic Heading*:

```
[heading Heading]
```

The *Generic Heading* assumes the level, plus one, of the innermost section where it is placed. For example, if it is placed in the outermost section, then, it assumes *h2*.

Headings are often used as an alternative to sections. It is used particularly if you do not want to start a new section. In many cases, however, headings in a particular section is just flat. Example:

```
[section A]
[h2 X]
[h2:link_id Y]
[h2 Z]
[endsect]
```

Here we use h2 assuming that section A is the outermost level. If it is placed in an inner level, you'll have to use h3, h4, etc. depending on where the section is. In general, it is the section level plus one. It is rather tedious, however, to scan the section level everytime. If you rewrite the example above as shown below, this will be automatic:

```
[section A]
[heading X]
[heading Y]
[heading Z]
[endsect]
```

They work well regardless where you place them. You can rearrange sections at will without any extra work to ensure correct heading levels. In fact, with *section* and *heading*, you have all you need. *h1..h6* becomes redundant. *h1..h6* might be deprecated in the future.

Macros

```
[def macro_identifier some text]
```

When a macro is defined, the identifier replaces the text anywhere in the file, in paragraphs, in markups, etc. macro_identifier is a string of non- white space characters except ']'. A macro may not follow an alphabetic character or the underscore. The replacement text can be any phrase (even marked up). Example:

```
[def sf_logo [$http://sourceforge.net/sflogo.php?group_id=28447&amp;type=1]]
sf_logo
```

Now everywhere the sf_logo is placed, the picture will be inlined.

Tip

It's a good idea to use macro identifiers that are distinguishable. For instance, in this document, macro identifiers have two leading and trailing underscores (e.g. `__spirit__`). The reason is to avoid unwanted macro replacement.

Links (URLS) and images are good candidates for macros. **1**) They tend to change a lot. It is a good idea to place all links and images in one place near the top to make it easy to make changes. **2**) The syntax is not pretty. It's easier to read and write, e.g. `__spirit__` than `[@http://spirit.sourceforge.net Spirit]`.

Some more examples:

```
[def :-)            [$theme/smiley.png]]
[def __spirit__     [@http://spirit.sourceforge.net Spirit]]
```

(See Images and Links)

Invoking these macros:

```
Hi __spirit__  :-)
```

will generate this:

Hi Spirit 

Predefined Macros

Quickbook has some predefined macros that you can already use.

Table 3. Predefined Macros

Macro	Meaning	Example
`__DATE__`	Today's date	2014-Aug-14
`__TIME__`	The current time	02:26:34 PM
`__FILENAME__`	Quickbook source filename	block.qbk

Templates

Templates provide a more versatile text substitution mechanism. Templates come in handy when you need to create parameterizable, multi-line, boilerplate text that you specify once and expand many times. Templates accept one or more arguments. These arguments act like place-holders for text replacement. Unlike simple macros, which are limited to phrase level markup, templates can contain block level markup (e.g. paragraphs, code blocks and tables).

Example template:

```
[template person[name age what]

Hi, my name is [name]. I am [age] years old. I am a [what].

]
```

Template Identifier

Template identifiers can either consist of:

- An initial alphabetic character or the underscore, followed by zero or more alphanumeric characters or the underscore. This is similar to your typical C/C++ identifier.
- A single character punctuation (a non-alphanumeric printable character)

Formal Template Arguments

Template formal arguments are identifiers consisting of an initial alphabetic character or the underscore, followed by zero or more alphanumeric characters or the underscore. This is similar to your typical C/C++ identifier.

A template formal argument temporarily hides a template of the same name at the point where the template is expanded. Note that the body of the `person` template above refers to `name` `age` and `what` as `[name]` `[age]` and `[what]`. `name` `age` and `what` are actually templates that exist in the duration of the template call.

Template Body

The template body can be just about any QuickBook block or phrase. There are actually two forms. Templates may be phrase or block level. Phrase templates are of the form:

```
[template sample[arg1 arg2...argN] replacement text... ]
```

Block templates are of the form:

```
[template sample[arg1 arg2...argN]
replacement text...
]
```

The basic rule is as follows: if a newline immediately follows the argument list, then it is a block template, otherwise, it is a phrase template. Phrase templates are typically expanded as part of phrases. Like macros, block level elements are not allowed in phrase templates.

Template Expansion

You expand a template this way:

```
[template_identifier arg1..arg2..arg3]
```

At template expansion, you supply the actual arguments. The template will be expanded with your supplied arguments. Example:

```
[person James Bond..39..Spy]
[person Santa Clause..87..Big Red Fatso]
```

Which will expand to:

Hi, my name is James Bond. I am 39 years old. I am a Spy.

Hi, my name is Santa Clause. I am 87 years old. I am a Big Red Fatso.

Caution

A word of caution: Templates are recursive. A template can call another template or even itself, directly or indirectly. There are no control structures in QuickBook (yet) so this will always mean infinite recursion. QuickBook can detect this situation and report an error if recursion exceeds a certain limit.

Each actual argument can be a word, a text fragment or just about any QuickBook phrase. Arguments are separated by the double dot "`..`" and terminated by the close parenthesis.

Note that templates and template parameters can't be expanded everywhere, only where text is interpreted as a phrase. So they can't be expanded in places such as table titles and link's urls. If you want to use a template to generate a link based of the template parameter, you can't use a normal link and will need to use escaped docbook instead. Example:

```
[template boost_ticket[key] '''<ulink url="https://svn.boost.org/trac/boost/tick↵
et/'''[key]'''">#'''[key]'''</ulink>''']

[boost_ticket 2035]
```

will expand to:

#2035

Caution

Since quickbook doesn't understand the context where the parameter is being used, it will interpret it as quickbook markup, so when writing a template like this, you'll need to escape any meaningful punctuation.

Nullary Templates

Nullary templates look and act like simple macros. Example:

```
[template alpha[]&apos;&apos;&apos;&amp;#945;&apos;&apos;&apos;]
[template beta[]&apos;&apos;&apos;&amp;#946;&apos;&apos;&apos;]
```

Expanding:

```
Some squigles...[*[alpha][beta]]
```

We have:

Some squiggles...αβ

The difference with macros are

- The explicit template expansion syntax. This is an advantage because, now, we don't have to use obscure naming conventions like double underscores (e.g. __alpha__) to avoid unwanted macro replacement.
- The template is expanded at the point where it is invoked. A macro is expanded immediately at its point of declaration. This is subtle and can cause a slight difference in behavior especially if you refer to other macros and templates in the body.

The empty brackets after the template identifier (`alpha[]`) indicates no arguments. If the template body does not look like a template argument list, we can elide the empty brackets. Example:

```
[template aristotle_quote Aristotle: [*['Education is the best provision
for the journey to old age.]]]
```

Expanding:

```
Here's a quote from [aristotle_quote].
```

We have:

Here's a quote from Aristotle: **Education is the best provision for the journey to old age.**.

The disadvantage is that you can't avoid the space between the template identifier, `aristotle_quote`, and the template body "Aristotle...". This space will be part of the template body. If that space is unwanted, use empty brackets or use the space escape: "`\ `". Example:

```
[template tag\ _tag]
```

Then expanding:

```
`struct` x[tag];
```

We have:

`struct` x_tag;

You have a couple of ways to do it. I personally prefer the explicit empty brackets, though.

Simple Arguments

As mentioned, arguments are separated by the double dot "`..`". Alternatively, if the double dot isn't used and more than one argument is expected, QuickBook uses whitespace to separate the arguments, following this logic:

- Break the last argument into two, at the first space found (`'', '\n', \t' or '\r'`).
- Repeat until there are enough arguments or if there are no more spaces found (in which case, an error is reported).

For example:

```
[template simple[a b c d] [a][b][c][d]]
[simple w x y z]
```

will produce:

wxyz

"w x y z" is initially treated as a single argument because we didn't supply any "`..`" separators. However, since `simple` expects 4 arguments, "w x y z" is broken down iteratively (applying the logic above) until we have "w", "x", "y" and "z".

QuickBook only tries to get the arguments it needs. For example:

```
[simple w x y z trail]
```

will produce:

wxyz trail

The arguments being: "w", "x", "y" and "z trail".

Caution

The behavior described here is for QuickBook 1.5. In older versions you could use both the double dot and whitespace as separators in the same template call. If your document is marked up as an older version, it will use the old behavior, which is described in the QuickBook 1.4 documentation.

Punctuation Templates

With templates, one of our objectives is to allow us to rewrite QuickBook in QuickBook (as a qbk library). For that to happen, we need to accommodate single character punctuation templates which are fairly common in QuickBook. You might have noticed that single character punctuations are allowed as template identifiers. Example:

```
[template ![bar] <hey>[bar]</hey>]
```

Now, expanding this:

```
[!baz]
```

We will have:

```
<hey>baz</hey>
```

Blurbs

```
[blurb :-) [*An eye catching advertisement or note...]

    Spirit is an object-oriented recursive-descent parser generator framework
    implemented using template meta-programming techniques. Expression templates
    allow us to approximate the syntax of Extended Backus-Normal Form (EBNF)
    completely in C++.
]
```

will generate this:

An eye catching advertisement or note...

Spirit is an object-oriented recursive-descent parser generator framework implemented using template meta-programming techniques. Expression templates allow us to approximate the syntax of Extended Backus-Normal Form (EBNF) completely in C++.

Note

Prefer admonitions wherever appropriate.

Tables

```
[table:id A Simple Table
    [[Heading 1] [Heading 2] [Heading 3]]
    [[R0-C0]     [R0-C1]     [R0-C2]]
    [[R1-C0]     [R1-C1]     [R1-C2]]
    [[R2-C0]     [R2-C1]     [R2-C2]]
]
```

will generate:

Table 4. A Simple Table

Heading 1	Heading 2	Heading 3
R0-C0	R0-C1	R0-C2
R1-C0	R1-C1	R1-C2
R2-C0	R2-C1	R2-C2

The table title is optional. The first row of the table is automatically treated as the table header; that is, it is wrapped in `<thead>...</thead>` XML tags. Note that unlike the original QuickDoc, the columns are nested in [cells...].

Giving tables an id is a new feature for quickbook 1.5 onwards. As with sections, the id is optional. If the table has a title but no id, an id will be generated from the title. The table above can be linked to using:

```
[link quickbook.syntax.block.tables.id link to table]
```

which will generate:

link to table

The syntax is free-format and allows big cells to be formatted nicely. Example:

```
[table Table with fat cells
    [[Heading 1] [Heading 2]]
    [
        [Row 0, Col 0: a small cell]
        [
            Row 0, Col 1: a big fat cell with paragraphs

            Boost provides free peer-reviewed portable C++ source libraries.

            We emphasize libraries that work well with the C++ Standard Library.
            Boost libraries are intended to be widely useful, and usable across
            a broad spectrum of applications. The Boost license encourages both
            commercial and non-commercial use.
        ]
    ]
    [
        [Row 1, Col 0: a small cell]
        [Row 1, Col 1: a small cell]
    ]
]
```

and thus:

Table 5. Table with fat cells

Heading 1	Heading 2
Row 0, Col 0: a small cell	Row 0, Col 1: a big fat cell with paragraphs Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.
Row 1, Col 0: a small cell	Row 1, Col 1: a small cell

Here's how to have preformatted blocks of code in a table cell:

```
[table Table with code
    [[Comment] [Code]]
    [
        [My first program]
        [``
            #include <iostream>

            int main()
            {
                std::cout << "Hello, World!" << std::endl;
                return 0;
            }
        ``]
    ]
]
```

Table 6. Table with code

Comment	Code

My first program	`#include <iostream>`
	
	`int main()`
	`{`
	`  std::cout << "Hello, World!" << std::endl;`
	`  return 0;`
	`}`

Variable Lists

```
[variablelist A Variable List
    [[term 1] [The definition of term 1]]
    [[term 2] [The definition of term 2]]
    [[term 3] [
    The definition of term 3.

    Definitions may contain paragraphs.
    ]]
]
```

will generate:

A Variable List

term 1 The definition of term 1

term 2 The definition of term 2

term 3 The definition of term 3.

Definitions may contain paragraphs.

The rules for variable lists are the same as for tables, except that only 2 "columns" are allowed. The first column contains the terms, and the second column contains the definitions. Those familiar with HTML will recognize this as a "definition list".

Include

You can include one QuickBook file from another. The syntax is simply:

```
[include someother.qbk]
```

In quickbook 1.6 and later, if the included file has a docinfo block then it will create a nested document. This will be processed as a standalone document, although any macros or templates from the enclosing file will still be defined.

Otherwise the included file will be processed as if it had been cut and pasted into the current document, with the following exceptions:

- The __FILENAME__ predefined macro will reflect the name of the file currently being processed.
- Any macros or templates defined in the included file are scoped to that file, i.e. they are not added to the enclosing file.

Note

In quickbook 1.5 and earlier templates weren't scoped in included files. If you want to use templates or macros from a file in quickbook 1.6, use import instead.

The `[include]` directive lets you specify a document id to use for the included file. You can specify the id like this:

```
[include:someid someother.qbk]
```

All auto-generated anchors will use the document id as a unique prefix. So for instance, if there is a top section in someother.qbk named "Intro", the named anchor for that section will be "someid.intro", and you can link to it with `[link someid.intro The Intro]`.

If the included file has a docinfo block, an id specified in an `[include]` directive will overwrite it.

You can also include C, C++ and python source files. This will include any quickbook blocks in the file that aren't inside of named code snippets. See the Import section for syntax details. For example, say you included this file:

```
/**
 * Hello world example
 */

// In this comment, the backtick indicates that this is a
// quickbook source block that will be included.

/*`
First include the appropriate header: [hello_includes]
Then write your main function: [hello_main]
*/

// This defines a code snippet, the syntax is
// described in the import section. It's available
// in the whole of this source file, not just after
// its definition.

//[hello_includes
#include <iostream>
//]

//[hello_main
int main() {
    std::cout << "Hello, trivial example" << std::endl;
}
//]
```

It will generate:

```
First include the appropriate header:

    #include <iostream>

Then write your main function:

    int main() {
        std::cout << "Hello, trivial example" << std::endl;
    }
```

Import

In quickbook 1.6 and later if you wish to use a template, macro or code snippet from a file, you need to import it. This will not include any of the content from that file, but will pull templates, macros and code snippets into the current file's scope.

With quickbook files, this allows you to create template and macro libraries. For python (indicated by the `.py` extension), C or C++ files this allows you to include code snippets from source files, so that your code examples can be kept up to date and fully tested.

Example

You can effortlessly import code snippets from source code into your QuickBook. The following illustrates how this is done:

```
[import ../test/stub.cpp]
[foo]
[bar]
```

The first line:

```
[import ../test/stub.cpp]
```

collects specially marked-up code snippets from stub.cpp and places them in your QuickBook file as virtual templates. Each of the specially marked-up code snippets has a name (e.g. `foo` and `bar` in the example above). This shall be the template identifier for that particular code snippet. The second and third line above does the actual template expansion:

```
[foo]
[bar]
```

And the result is:

This is the **foo** function.

This description can have paragraphs...

- lists
- etc.

And any quickbook block markup.

```
std::string foo()
{
    // return 'em, foo man!
    return "foo";
}
```

This is the **bar** function

```
std::string bar()
{
    // return 'em, bar man!
    return "bar";
}
```

Some trailing text here

Code Snippet Markup

Note how the code snippets in stub.cpp get marked up. We use distinguishable comments following the form:

```
//[id
some code here
//]
```

The first comment line above initiates a named code-snippet. This prefix will not be visible in quickbook. The entire code-snippet in between `//[id` and `//]` will be inserted as a template in quickbook with name id. The comment `//]` ends a code-snippet This too will not be visible in quickbook.

Special Comments

Special comments of the form:

```
//` some [*quickbook] markup here
```

and:

```
/*` some [*quickbook] markup here */
```

will be parsed by QuickBook. This can contain quickbook *blocks* (e.g. sections, paragraphs, tables, etc). In the first case, the initial slash-slash, tick and white-space shall be ignored. In the second, the initial slash-star-tick and the final star-slash shall be ignored.

Special comments of the form:

```
/*<- this C++ comment will be ignored ->*/
```

or

```
/*<-*/ "this c++ code  will be ignored" /*->*/
```

or

```
//<-
private:
    int some_member;
//->
```

can be used to inhibit code from passing through to quickbook. All text between the delimeters will simply be ignored.

Comments of this form:

```
//=int main() {}
```

or

```
/*=foo()*/
```

will be displayed as code that isn't in comments. This allows you to include some code in the snippet but not actually use it when compiling your example.

Callouts

Special comments of the form:

```
/*< some [*quickbook] markup here >*/
```

will be regarded as callouts. These will be collected, numbered and rendered as a "callout bug" (a small icon with a number). After the whole snippet is parsed, the callout list is generated. See Callouts for details. Example:

```
std::string foo_bar() ❶
{
    return "foo-bar"; ❷
}
```

❶ The *Mythical* FooBar. See Foobar for details
❷ return 'em, foo-bar man!

This is the actual code:

```
//[ foo_bar
std::string foo_bar() /*< The /Mythical/ FooBar.
                         See [@http://en.wikipedia.org/wiki/Foobar Foobar for details] >*/
{
    return "foo-bar"; /*< return 'em, foo-bar man! >*/
}
//]
```

The callouts bugs are placed exactly where the special callout comment is situated. It can be anywhere in the code. The bugs can be rather obtrusive, however. They get in the way of the clarity of the code. Another special callout comment style is available:

```
/*<< some [*quickbook] markup here >>*/
```

This is the line-oriented version of the callout. With this, the "bug" is placed at the very left of the code block, away from the actual code. By placing it at the far left, the code is rendered un-obscured. Example:

```
class x
{
public:

    ❶x() : n(0)
    {
    }

    ❷~x()
    {
    }

    ❸int get() const
    {
        return n;
    }

    ❹void set(int n_)
    {
        n = n_;
    }
};
```

❶ Constructor
❷ Destructor
❸ Get the n member variable
❹ Set the n member variable

See the actual code here: tools/quickbook/test/stub.cpp

Plain blocks

`block` is a plain block element, that doesn't wrap its contents in any docbook or boostbook tags. This can be useful when using escaped docbook block tags, such as:

```
[template chapter[title]
[block'''<chapter><title>'''[title]'''</title>''']
]

[template chapterend
[block'''</chapter>''']
]

[chapter An example chapter]

Content

[chapterend]
```

Without the `block` element, the `chapter` and `chapterend` templates would be wrapped in paragraph tags.

Note

In this example, the template body has to start with a newline so that the template will be interpreted in block mode.

Language Versions

Stable Versions

Since quickbook 1.3 the `quickbook` attribute in the document block selects which version of the language to use. Not all changes to quickbook are implemented using a version switch, it's mainly just the changes that change the way a document is interpreted or would break existing documentation.

Quickbook 1.3 and later

- Introduced quickbook language versioning.
- In the documentation info, allow phrase markup in license and purpose attributes.
- Fully qualified section and headers. Subsection names are concatenated to the ID to avoid clashing. Example: `doc_name.sect_name.sub_sect_name.sub_sub_sect_name`.

Quickbook 1.5 and later

- Ignore template argument separators inside square brackets.
- Don't separate the final template argument if the `..` separator was used. i.e. never mix `..` and whitespace separators.
- Statically scope templates and their arguments rather than dynamically scope them.
- Give table ids, and let you set them.
- Allow spaces between the `:` character and ids in elements which can have ids.

Quickbook 1.6

Upgrading a document from an earlier version of quickbook shouldn't be too hard. The first thing to do is update the version in the docinfo block. For example, if you were updating the Xpressive documentation, the existing docinfo block looks like:

```
[library Boost.Xpressive
    [quickbook 1.3]
    ...
]
```

Change this to:

```
[library Boost.Xpressive
    [quickbook 1.6]
    [compatibility-mode 1.3]
    ...
]
```

The `compatibility-mode` tag ensures that automatically generated links won't change. It might turn out that it isn't required, but in Xpressive's case if it isn't included it will break a lot of links.

Then try building it. You might need to fix some stray square brackets. The new version has a stricter parser which will have an error for brackets which don't pair up. They might be there by mistake, in which case they should probably be deleted, or if they're intentional escaped. For example, to write out the half-open range [a,b), use: `\[a,b)`.

Next, you might need to reconsider how templates and macros are defined. If you `include` a file to use its templates, you'll now need to `import` it instead as templates are now scoped by included files. Also, if you define templates and macros in your main quickbook file, you might want to put them into a separate file and `import` that, which allows the main documentation files to concentrate on the structure and contents of the document, making them easier to read.

Now that headings can have ids, it can be a good idea to add ids to existing headings. This means that the headings will have more predictable ids which don't change when the text of the heading changes. In order to preserve links you can use the existing generated id as the heading.

Includes with docinfo

In quickbook 1.5 if you include a file which starts with a docinfo block, it's ignored and the file is expanded in place. In quickbook 1.6 it's treated as a document nested in the current position. So if it has an 'article' docinfo block, boostbook 'article' tags are used.

It also mostly generates the same markup as if the file was converted separately - so for example, the same ids are generated, the document is processed using the language version specified in the docinfo block. If no language is specified it uses the default (1.1) not the version of the document that included it. This might seem surprising, but is requried so that quickbook will convert it the same way as if it was converted separately.

So for the most part, includes with a docinfo are like an `xinclude`, apart from a couple of differences. Templates and macros defined in the parent document are used in the included document, and the id generator rewrites ids that clash between multiple documents.

If an included document doesn't have a docinfo block, it's just included as before.

Macros in docinfo block

You can now expand macros in text fields in the docinfo block. In the top docinfo block only the predefined macros are available, but in nested documents macros defined in the parent document are also available.

There's a small bug here - this leaks into older versions for the `license` and `purpose` fields, but since only the predefined macros are available, it's unlikely to break any existing documents. So I'd rather not complicate the code further by fixing that.

Scoping templates and macros

A long standing quickbook bug is that macros are scoped by file, but templates aren't. So you can define templates in a separate file and include them, but not macros. This has been fixed so that templates defined in one file won't 'leak' into another.

But this means there's no way to define templates in a separate file - a useful feature. To do this the `import` element has been adapted to also support quickbook files. If a quickbook file is imported, the templates and macros defined in it are added to the current scope, but nothing else contained in that file is used. This could be used to create template and macro library files. This matches the existing semantics of importing code snippets.

When importing templates, they're bound to the language version of the file they were defined in. This means that if you import them into a file with a different version it won't change the way they're interpreted. Although, as we'll see later, the generated boostbook is slightly different.

Including C++ and python files

As `import` now supports quickbook files, `include` also supports source files. It includes any quickbook contained in comments outside of code snippets. Code snippets in the file are available to be expanded within the file but are scoped to the file. In exactly the same manner as when templates and macros are scoped in an included quickbook file.

Id Generation

Id generation in quickbook 1.5 is a bit buggy, but that can't be fixed without a version switch as it will break existing documents. For example in quickbook 1.5 when you include a quickbook file, it stops using the explicit id from the documentation info and generates a new id from the document title to use instead.

The id generator in quickbook 1.6 has been improved in some other ways to. When generating ids from section titles, table titles etc. it always uses the quickbook source rather than the generated boostbook to generate the id. It then cleans up the id slightly, trimming leading and trailing underscores and replacing multiple underscores with a single underscore. Then if the newly generated part of the id is longer than 32 characters it truncates it.

While the new id generator generally creates better ids, it's more likely to generate duplicates so quickbook needs to handle duplicates better. When there are multiple identical ids, quickbook chooses one to use based on a priority list - anchors are preferred, then explicit document and section ids, then other explicit ids, followed by the generated ids. Then any other explicit ids in the document have numbers added to avoid duplicates - first the explicit ids in the order they appear and then the generated ids. A generated id which accidentally clashes with an explicit id should never change the explicit id.

Older language versions still generate the same ids they always have, with the exception of duplicate ids which are handled using the new mechanism - this is not a breaking change since duplicate ids can't be linked to.

Compatibility Mode

As mentioned before, changing the id generator will break links in documents written using an old language version. So to ease the transition a 'compatibility mode' is used, this just requires an extra attribute in the docinfo, for example if you're converting a 1.5 document to 1.6:

```
[article Document
[quickbook 1.6]
[compatibility-mode 1.5]
]
```

This means the document will be parsed as 1.6, using all the new features, but ids (and possibly other markup) will generated as they were for a 1.5 document.

Compatibility mode is also implicitly used when generating templates written in a different language version to the current document. So the template is parsed in the version it was written for, but generates boostbook that's compatible with the current document.

Version info outside of document info block

Can now use `quickbook` and `compatibility-mode` tags at the beginning of the file. Either before or without a document info block. This is useful for files just containing templates, which don't really need a document info block.

If you don't specify `compatibility-mode`, the behaviour depends on whether or not you have a docinfo block. If you do it uses the file's quickbook version, if you don't it inherits the parent's compatibility mode even if you specify a quickbook version. This is the right thing to do - mixing compatibility modes within documents is problematic. It might actually be a mistake to allow them to specified outside docinfo blocks.

This change is also backdated to older versions. So when including from an older version, the included file's version can be set (older versions ignore document info in included files).

Explicit Heading Ids

Headings can now be given explicit ids:

```
[heading:id A heading with an explicit id]
```

Punctuation changes

In 1.6, quickbook is more consistent about how it parses punctuation. Escapes are now supported in links, anchors, table titles, image attributes etc. The flip side of this is that quickbook is now stricter about unescaped brackets. They can still be used, but need to match up, otherwise there's an error.

Since quickbook now matches up square brackets it will fix some mis-parses. For example `[*[bold]]` used to parse as **[bold**] - note that the closing square bracket isn't bold, now it parses as **[bold]**. In this case it's just a subtle visual difference, but it could cause odd problems, for example when nested in a table cell.

Table Titles

Table titles are now parsed as phrases, so some markup is allowd:

```
[table [*bold title]]
```

Which is an empty table with a bold title. The title is no longer ended by a newline, but by either a closing square bracket, or two opening square brackets - which you get at the start of the table cells, so this now works:

```
[table Simple[[heading 1][heading 2]][[cell 1][cell 2]]]
```

XML base

A problem when using `xi:include` tags in escaped boostbook is that you typically don't know which directory the boostbook file will be in, so it's impossible to use relative links. This can be fixed by adding an `xml:base` attribute to the document tag. To do this use the new `xmlbase` attribute in your document's docinfo block. For example to make escaped `xi:includes` be relative to the directory of the file:

```
[library Library documentation
[quickbook 1.6]
[xmlbase .]
]
```

Any paths in `xinclude` elements will be rewritten accordingly. Note that most documents won't need this, and probably shouldn't use it. Only use it if you're totally sure that you will need it.

Improved template parser

There's a new parser for template declarations and parameters which does a better job of understanding escaped and bracketed text. Unfortunately it does not understand element names so there are some cases where it could go wrong. For example:

```
[template doesnt_work[]
[ordered_list
    [`code phrase`]
]
]
```

In this case it will think the `[\`` is a template call and give a parse error. To work around this put an escaped space before the code phrase:

```
[template works[]
[ordered_list
    [\ `code phrase`]
]
]
```

New Elements

New elements added in quickbook 1.6:

- `block`
- `ordered_list` and `itemized_list`
- `role`

Quickbook 1.7

Error for elements used in incorrect context

Previously if you used an element in the wrong context it would just be unprocessed, which was surprising. People often didn't realise that their element hadn't been processed. So now it's an error.

Error for invalid phrase elements

If the body of a phrase element didn't parse, it would be just used unprocessed. Now change it to be a hard error.

Source mode for single entities

1.7 introduces a new `!` element type for setting the source mode of a single entity without changing the source mode otherwise. This can be used for code blocks and other elements. For example:

```
[!c++]
    void foo() {};

[!python]```def foo():```
```

It can also be used to set the source mode for elements:

```
[!teletype][table
    [[code][meaning]]
    [[`+`][addition]]
]
```

When used before a section, it sets the source mode for the whole section.

If it appears at the beginning of a paragraph, it will be used for the whole paragraph only if there's a newline, eg.

```
[!c++]
A declaration `void foo();` and a definition `void foo() {}`.
```

Callouts in code blocks

Currently callouts can only be used in code snippets. 1.7 adds support in normal code blocks. The same syntax is used as in code snippets, the callout descriptions appear immediately after the code block.

Escaped docbook in docinfo blocks

Quickbook docinfo attributes will probably never be as rich as docbook attributes. To allow more flexible markup that is not supported by quickbook, escaped docbook can be included in the docinfo block:

```
[article Some article
[quickbook 1.7]
'''<author>
    <firstname>John</firstname>
    <surname>Doe</surname>
    <email>john.doe@example.com</email>
</author>'''
]
```

The escaped docbook is always placed at the end of the docinfo block, so it shouldn't be assumed that it will interleave with markup generated from quickbook. A mixture of quickbook and docbook attributes for the same information will not work well.

Pargraphs in lists

Paragraphs and block elements can now be used in lists:

```
* Para 1

  Para 2
  * Nested Para 1

    Nested Para 2

        Code block
  Para 3
```

generates:

- Para 1

  Para 2

  - Nested Para 1

    Nested Para 2

    ```
    Code block
    ```

  Para 3

Templates in some attributes

There's support for calling templates in link values, anchors, roles and includes. This is sometimes a bit of a change, especially in places where spaces are currently allowed, so I might try using a slightly different grammar where required. I think I also need to add some validation, since the parser can allow more symbols than some of the old ones.

List Markup in Nested Blocks

Can now place list markup in nested blocks, e.g in tables, variables lists etc. Unfortunately indented code blocks are more tricky, because the contents of these blocks are often indented already. It seemed easier to just not support indented code blocks in this context than to try to work out sensible actions for the edges cases. If you want to use code blocks in this context, you should still be able to use explicit markup.

Allow block elements in phrase templates

Block elements can now be used in phrase templates, but paragraphs breaks aren't allowed, so this is an error:

```
[template paras[] Something or other.

Second paragraph.]
```

If a phrase template only contains block elements, then it's practically indistinguishable from a block template. So you'll get the same output from:

```
[template foo[] [blurb Blah, blah, blah]]
```

as:

```
[template foo[]
[blurb Blah, blah, blah]
]
```

If a phrase template has phrase content mixed with block elements, it'll generate output as if it was expanded inline.

Including multiple files with Globs

One can now include multiple files at once using a glob pattern for the file reference:

```
[include sub/*/*.qbk]
[include include/*.h]
```

All the matching files, and intermediate irectories, will match and be included. The glob pattern can be "*" for matching zero or more characters, "?" for matching a single character, "[<c>-<c>]" to match a character class, "[^<char>-<char>]" to exclusive match a character class, "\\" to escape a glob special character which is then matched, and anything else is matched to the character.

Note

Because of the escaping in file references the "\\" glob escape is a double "\"; i.e. and escaped back-slash.

Installation and configuration

This section provides some guidelines on how to install and configure BoostBook and Quickbook under several operating systems. Before installing you'll need a local copy of boost, and to install the version of `bjam` which comes with it (or a later version).

Mac OS X

The simplest way to install on OS X is to use macports. If you don't want to use macports and are using Snow Leopard or later, there are instructions later. Earlier versions of OS X need to use something like macports to install `xsltproc` because the version they come with is very old, and doesn't have good enough XSL support for boostbook's stylesheets.

Mac OS X, using macports

First install the `libxslt`, `docbook-xsl` and `docbook-xml-4.2` packages:

```
sudo port install libxslt docbook-xsl docbook-xml-4.2
```

Next, we need to configure Boost Build to compile BoostBook files. Add the following to your `user-config.jam` file, which should be in your home directory. If you don't have one, create a file containing this text. For more information on setting up `user-config.jam`, see the Boost Build documentation.

```
using xsltproc
    : /opt/local/bin/xsltproc
    ;

using boostbook
    : /opt/local/share/xsl/docbook-xsl/
    : /opt/local/share/xml/docbook/4.2
    ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds:

1. Go to Quickbook's source directory (`BOOST_ROOT/tools/quickbook`).
2. Build the utility by issuing `bjam`.
3. Copy the resulting `quickbook` binary (located at `BOOST_ROOT/dist/bin`) to a safe place. The traditional location is `/usr/local/bin`.
4. Add the following to your `user-config.jam` file, using the full path of the quickbook executable:

   ```
   using quickbook
       : /usr/local/bin/quickbook
       ;
   ```

If you need to build documentation that uses Doxygen, you will need to install it as well:

```
sudo port install doxygen
```

And then add to your `user-config.jam`:

```
using doxygen ;
```

Alternatively, you can install from the official doxygen `dmg`. This is described at the end of the next section.

Mac OS X, Snow Leopard (or later)

Section contributed by Julio M. Merino Vidal

The text below assumes you want to install all the necessary utilities in a system-wide location, allowing any user in the machine to have access to them. Therefore, all files will be put in the `/usr/local` hierarchy. If you do not want this, you can choose any other prefix such as `~/Applications` for a single-user installation.

Snow Leopard comes with `xsltproc` and all related libraries preinstalled, so you do not need to take any extra steps to set them up. It is probable that future versions will include them too, but these instructions may not apply to older versions.

To get started:

1. Download Docbook XML 4.2 and unpack it inside `/usr/local/share/xml/docbook/4.2`.

2. Download the latest Docbook XSL version and unpack it. Be careful that you download the correct file, sometimes the 'looking for the latest version' link often links to another file. The name should be of the form `docbook-xsl-1.`*`nn.n`*`.tar.bz2`, with no suffix such as `-ns.tar.bz2` or `-doc.tar.bz2`. Put the results in `/usr/local/share/xsl/docbook`, thus effectively removing the version number from the directory name (for simplicity).

3. Add the following to your `user-config.jam` file, which should live in your home directory (`/Users/<your_username>`). You must already have it somewhere or otherwise you could not be building Boost (i.e. missing tools configuration).

```
using xsltproc ;

using boostbook
    : "/usr/local/share/xsl/docbook"
    : "/usr/local/share/xml/docbook/4.2"
    ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds and install a system-wide Quickbook instead:

1. Go to Quickbook's source directory (`BOOST_ROOT/tools/quickbook`).

2. Build the utility by issuing `bjam`.

3. Copy the resulting `quickbook` binary (located at `BOOST_ROOT/dist/bin`) to a safe place. Following our previous example, you can install it into: `/usr/local/bin`.

4. Add the following to your `user-config.jam` file:

```
using quickbook
    : "/usr/local/bin/quickbook" ;
    ;
```

Additionally, if you need to build documentation that uses Doxygen, you will need to install it too:

1. Go to the downloads section and get the disk image (`dmg` file) for Mac OS X.

2. Open the disk image and drag the Doxygen application to your `Applications` folder to install it.

3. Add the following to your `user-config.jam` file:

```
using doxygen
    : /Applications/Doxygen.app/Contents/Resources/doxygen
    ;
```

Windows 2000, XP, 2003, Vista, 7

Section contributed by Julio M. Merino Vidal

The following instructions apply to any Windows system based on Windows 2000, including Windows XP, Windows 2003 Server, Windows Vista, and Windows 7. The paths shown below are taken from a Windows Vista machine; you will need to adjust them to match your system in case you are running an older version.

1. First of all you need to have a copy of `xsltproc` for Windows. There are many ways to get this tool, but to keep things simple, use the binary packages made by Igor Zlatkovic. At the very least, you need to download the following packages: `iconv`, `zlib`, `libxml2` and `libxslt`.

2. Unpack all these packages in the same directory so that you get unique `bin`, `include` and `lib` directories within the hierarchy. These instructions use `C:\Users\example\Documents\boost\xml` as the root for all files.

3. From the command line, go to the `bin` directory and launch `xsltproc.exe` to ensure it works. You should get usage information on screen.

4. Download Docbook XML 4.2 and unpack it in the same directory used above. That is: `C:\Users\example\Documents\boost\xml\docbook-xml`.

5. Download the latest Docbook XSL version and unpack it, again in the same directory used before. Be careful that you download the correct file, sometimes the 'looking for the latest version' link often links to another file. The name should be of the form `docbook-xsl-1.`*`nn.n`*`.tar.bz2`, with no suffix such as `-ns.tar.bz2` or `-doc.tar.bz2`. To make things easier, rename the directory created during the extraction to `docbook-xsl` (bypassing the version name): `C:\Users\example\Documents\boost\xml\docbook-xsl`.

6. Add the following to your `user-config.jam` file, which should live in your home directory (`%HOMEDRIVE%%HOMEPATH%`). You must already have it somewhere or otherwise you could not be building Boost (i.e. missing tools configuration).

```
using xsltproc
    : "C:/Users/example/Documents/boost/xml/bin/xsltproc.exe"
    ;

using boostbook
    : "C:/Users/example/Documents/boost/xml/docbook-xsl"
    : "C:/Users/example/Documents/boost/xml/docbook-xml"
    ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds:

1. Go to Quickbook's source directory (`BOOST_ROOT\tools\quickbook`).

2. Build the utility by issuing `bjam`.

3. Copy the resulting `quickbook.exe` binary (located at `BOOST_ROOT\dist\bin`) to a safe place. Following our previous example, you can install it into: `C:\Users\example\Documents\boost\xml\bin`.

4. Add the following to your `user-config.jam` file:

```
using quickbook
    : "C:/Users/example/Documents/boost/xml/bin/quickbook.exe"
    ;
```

Debian, Ubuntu

The following instructions apply to Debian and its derivatives. They are based on a Ubuntu Edgy install but should work on other Debian based systems. They assume you've already installed an appropriate version of `bjam` for your copy of boost.

First install the `xsltproc`, `docbook-xsl` and `docbook-xml` packages. For example, using `apt-get`:

```
sudo apt-get install xsltproc docbook-xsl docbook-xml
```

If you're planning on building boost's documentation, you'll also need to install the `doxygen` package as well.

Next, we need to configure Boost Build to compile BoostBook files. Add the following to your `user-config.jam` file, which should be in your home directory. If you don't have one, create a file containing this text. For more information on setting up `user-config.jam`, see the Boost Build documentation.

```
using xsltproc ;

using boostbook
    : /usr/share/xml/docbook/stylesheet/nwalsh
    : /usr/share/xml/docbook/schema/dtd/4.2
    ;

# Remove this line if you're not using doxygen
using doxygen ;
```

The above steps are enough to get a functional BoostBook setup. Quickbook will be automatically built when needed. If you want to avoid these rebuilds:

1. Go to Quickbook's source directory (`BOOST_ROOT/tools/quickbook`).
2. Build the utility by issuing `bjam`.
3. Copy the resulting `quickbook` binary (located at `BOOST_ROOT/dist/bin`) to a safe place. The traditional location is `/usr/local/bin`.
4. Add the following to your `user-config.jam` file, using the full path of the quickbook executable:

   ```
   using quickbook
       : /usr/local/bin/quickbook
       ;
   ```

Editor Support

Editing quickbook files is usually done with text editors both simple and powerful. The following sections list the settings for some editors which can help make editing quickbook files a bit easier.

Note

You may submit your settings, tips, and suggestions to the authors, or through the docs Boost Docs mailing list.

Scintilla Text Editor

Section contributed by Dean Michael Berris

The Scintilla Text Editor (SciTE) is a free source code editor for Win32 and X. It uses the SCIntilla source code editing component.

Tip

SciTE can be downloaded from http://www.scintilla.org/SciTE.html

You can use the following settings to highlight quickbook tags when editing quickbook files.

```
qbk=*.qbk
lexer.*.qbk=props
use.tabs.$(qbk)=0
tab.size.$(qbk)=4
indent.size.$(qbk)=4
style.props.32=$(font.base)
comment.stream.start.props=[/
comment.stream.end.props=]
comment.box.start.props=[/
comment.box.middle.props=
comment.box.end.props=]
```

Note

Thanks to Rene Rivera for the above SciTE settings.

KDE Support

boost::hs::quickbook

boost::hs::quickbook is a syntax highlighting designed to work with Katepart. It can be used in KWrite, Kate, Konqueror and KDevelop, and supports all the constructs of Quickbook 1.4 including tables, list, templates and macros.

.qbk loaded in a text editor

```
[table Code examples
[[    Name     ][                    Code                    ][          Description
[[for loop      ][ ` for(int k=0; k<10; k++) v+=k; `          ][Sums some numbers.
[[while loop    ][ ` { int k; while( k < 10 ) { v+=k; k++; } } ` ][Same effect.
[[infinite loop ][ ` while( true ) { v+=1; } `                ][Not a good example.
]
```

.qbk loaded with boost::hs support

```
[table Code examples
[[   Name     ][                    Code                    ][          Description
[[for loop    ][ ` for(int k=0; k<10; k++) v+=k; `          ][Sums some numbers.
[[while loop  ][ ` { int k; while( k < 10 ) { v+=k; k++; } } ` ][Same effect.
[[infinite loop ][ ` while( true ) { v+=1; } `              ][Not a good example.
]
```

html generated from this .qbk file

Table 7. Code examples

Name	Code	Description
for loop	`for(int k=0; k<10; k++) v+=k;`	Sums some numbers.
while loop	`{ int k; while( k < 10 ) { v+=k; k++ } }`	Same effect.
infinite loop	`while( true ) { v+=1; }`	Not a good example.

Code Folding

boost::hs goes far beyond simple coloring. One useful thing you can get the editor to do is to mark regions. They appear in a small grey line and each region can be folded or unfolded independently.

Auto Comment / Uncomment

Another important feature is the possibility to auto-comment or uncomment some piece of code (*Tools - Comment*). Commented regions can be uncommented simple calling the *uncomment* command while being in it.

Styles reference

Name	Style	Description
plain text	normal black	Plain text at each level.
formatted text	formatted black	Bold, italic, underline and mixes. Teletype, replaceable, strikeout.
structure	light blue	All quickbook structures characters ([,], [block-type, simple formating boundaries, lists keywords (*, #)
macros	red	Names in macro definitions, macros insertion if it is used the __xxx__ proposed sintaxis.
templates	red	Names in template definitions
anchors	red	All the keywords that are used to link quickbooks together.
comments	italic light gray	Inside the commentaries.
tables	HTML like	Reveal the structure, bold title, higlighted HTML like columns titles.
variable lists	HTML like	Reveal the structure, bold title, bold HTML like items names.
c++ code	cpp Kate syntax	Code blocks and inline code.
paths	green	Image, files and web paths
IDE specific	dark blue	IDE commands

About boost::hs

boost::hs::quickbook is a component of boost::hs, a syntax highlighting for C++, doxygen, Boost.Build jamfiles and QuickBook. boost::hs has his own page here.

Note

boost::hs::cpp support QuickBook code import comments style!

Installing boost::hs

There exist an ongoing effort to push boost::hs upstream to the KatePart project. In a few months KDE may have native Quickbook support! For the moment you must download and install it.

You can download boost::hs from here.

Note

A copy of boost::hs::quickbook and boost::hs::cpp is available in `boost/tools/quickbook/extra/katepart`.

In order to install it you must copy the content in the folder **katepart/syntax/** to the appropriate katepart syntax folder in your machine. In general this folder will be in **/usr/share/apps/katepart/syntax**. A bash script named *install.sh* is included that copy the files to this folder.

Frequently Asked Questions

Can I use QuickBook for non-Boost documentation?

QuickBook can be used for non-Boost documentation with a little extra work.

Faq contributed by Michael Marcin

When building HTML documentation with BoostBook a Boost C++ Libraries header is added to the files. When using QuickBook to document projects outside of Boost this is not desirable. This behavior can be overridden at the BoostBook level by specifying some XSLT options. When using Boost Build version 2 (BBv2) this can be achieved by adding parameters to the BoostBook target declaration.

For example:

```
using quickbook ;

xml my_doc : my_doc.qbk ;

boostbook standalone
    :
        my_doc
    :
        <xsl:param>boost.image.src=images/my_project_logo.png
        <xsl:param>boost.image.alt="\"My Project\""
        <xsl:param>boost.image.w=100
        <xsl:param>boost.image.h=50
        <xsl:param>nav.layout=none
    ;
```

Is there an easy way to convert BoostBook docs to QuickBook?

There's a stylesheet that allows Boostbook generated HTML to be viewed as quickbook source, see http://svn.boost.org/trac/boost/wiki/QuickbookSourceStylesheetProject, so it's then just a cut and paste job to convert the BoostBook to QuickBook (which IMO is a whole lot easier to edit and maintain).

--John Maddock

Quick Reference

Table 8. Syntax Compendium

To do this...	Use this...	See this...
comment	`[/ some comment]`	Comments
italics	`['italics] or /italics/`	Font Styles and Simple formatting
bold	`[*bold] or *bold*`	Font Styles and Simple formatting
underline	`[_underline] or _underline_`	Font Styles and Simple formatting
`teletype`	`[^teletype] or =teletype=`	Font Styles and Simple formatting
~~strikethrough~~	`[-strikethrough]`	Font Styles and Simple formatting
`replaceable`	`[~replaceable]`	Replaceble
source mode	`[c++]` or `[python]`	Source Mode
inline code	`` `int main();` ``	Inline code
code block	``` ``int main();`` ```	Code
code escape	``` ``from c++ to QuickBook`` ```	Escaping Back To QuickBook
line break	`[br] or \n`	line-break **DEPRECATED**
anchor	`[#anchor]`	Anchors
link	`[@http://www.boost.org Boost]`	Links
anchor link	`[link section.anchor Link text]`	Anchor links
refentry link	`[link xml.refentry Link text]`	refentry links
function link	`[funcref fully::qualified::function_name Link text]`	function, class, member, enum, macro, concept or header links
class link	`[classref fully::qualified::class_name Link text]`	function, class, member, enum, macro, concept or header links
member link	`[memberref fully::qualified::member_name Link text]`	function, class, member, enum, macro, concept or header links
enum link	`[enumref fully::qualified::enum_name Link text]`	function, class, member, enum, macro, concept or header links
macro link	`[macroref MACRO_NAME Link text]`	function, class, member, enum, macro, concept or header links
concept link	`[conceptref ConceptName Link text]`	function, class, member, enum, macro, concept or header links
header link	`[headerref path/to/header.hpp Link text]`	function, class, member, enum, macro, concept or header links

To do this...	Use this...	See this...
global link	`[globalref fully::qualified::global Link text]`	function, class, member, enum, macro, concept or header links
escape	`'''escaped text (no processing/formatting)'''`	Escape
single char escape	`\c`	Single char escape
images	`[$image.jpg]`	Images
begin section	`[section The Section Title]`	Section
end section	`[endsect]`	Section
paragraph	No markup. Paragraphs start left-flushed and are terminated by two or more newlines.	Paragraphs

ordered list	`# one`
	`# two`
	`# three`
		Ordered lists

unordered list	`* one`
	`* two`
	`* three`
		Unordered lists

code	No markup. Preformatted code starts with a space or a tab.	Code
preformatted	`[pre preformatted]`	Preformatted
block quote	`[:sometext...]`	Blockquote
heading 1	`[h1 Heading 1]`	Heading
heading 2	`[h2 Heading 2]`	Heading
heading 3	`[h3 Heading 3]`	Heading
heading 4	`[h4 Heading 4]`	Heading
heading 5	`[h5 Heading 5]`	Heading
heading 6	`[h6 Heading 6]`	Heading
macro	`[def macro_identifier some text]`	Macros
template	`[template[a b] [a] body [b]]`	Templates
blurb	`[blurb advertisement or note...]`	Blurbs

To do this...	Use this...	See this...
admonition	`[warning Warning text...]`	Admonitions

table	`[table Title`
	`[[a][b][c]]`
	`[[a][b][c]]`
	`]`
		Tables

variablelist	`[variablelist Title`
	`[[a][b]]`
	`[[a][b]]`
	`]`
		Variable Lists

include	`[include someother.qbk]`	Include
conditional generation	`[? symbol phrase]`	Conditional Generation