LinuxDoc-Tools User's Guide

written by Matt Welsh as the LinuxDoc-SGML User's Guide. Updated by Greg Hankins, and rewritten by Eric S. Raymond for SGML-Tools. Updated and renamed by Taketoshi Sano, for LinuxDoc-Tools $Date: 2002/03/18 13:39:10 $ ($Revision: 1.2 $)

This document is a user's guide to the LinuxDoc-Tools formatting system, a SGML-based system which allows you to produce a variety of output formats. You can create plain text output (ASCII, ISO-8859-1, and EUC-JP), DVI, PostScript, PDF, HTML, GNU info, LyX, and RTF output from a single document source file. LinuxDoc-Tools is a new branch from SGML-Tools 1.0.9, and an descendant of the original LinuxDoc-SGML.

Contents

1 Introduction

This document is the user's guide to the LinuxDoc-Tools document processing system. LinuxDoc-Tools is a suite of programs to help you write source documents that can be rendered as plain text, hypertext, or LaTeX files. It contains what you need to know to set up LinuxDoc-Tools and write documents using it. See `example.sgml` for an example of an LinuxDoc DTD SGML document that you can use as a model for your own documents. The "LinuxDoc" means the name of a specific SGML DTD here.

1.1 What's the DTD ?

The DTD specifies the names of "elements" within the document. An element is just a bit of structure; like a section, a subsection, a paragraph, or even something smaller like *emphasized text*. You may know the HTML has their own DTD.

Don't be confusing. SGML is *not* a text-formatting system. SGML itself is used only to specify the document structure. There are no text-formatting facilities or "macros" intrinsic to SGML itself. All of those things are defined within the DTD. You can't use SGML without a DTD; a DTD defines what SGML does. For more Detail, please refer the later section of this document (6 (How LinuxDoc-Tools Works)).

1.2 History of the LinuxDoc

The LinuxDoc DTD is created by Matt Welsh as the core part of his Linuxdoc-SGML document processing system. This DTD is based heavily on the QWERTZ DTD by Tom Gordon, `thomas.gordon@gmd.de`. The target of the QWERTZ DTD is to provide the simple way to create LaTeX source for document publishing. Matt Welsh took and shaped it into Linuxdoc-SGML because he needed it to produce a lot of Linux Documentations. It can convert a single source of documentation into various output formats such as plain text, html, and PS. No work for synchronization between various output formatted documents are needed.

The Linuxdoc-SGML system had been maintained for years by Matt Welsh and many others, but it has some limitations. Then Cees de Groot came and created the new system using perl. The new system is called as “SGML-Tools”. The perl based version for LinuxDoc had been maintained for a year, then totally new system using the original python scripts and some stylesheets with the jade has been released. This system is called as “SGML-Tools 2.0” and it does not use the LinuxDoc DTD as the main DTD, but uses the new standard one, the DocBook DTD. Now “SGML-Tools 2.0” becomes “SGMLtools-Lite” and is distributed from `http://sgmltools-lite.sourceforge.net/` .

Recently, the DocBook DTD is the standard DTD for the technical software documentation, and used by many project such as GNOME and KDE, as well as many professional authors and commercial publishers. But some people in the LDP, and users of the various LinuxDoc SGML documents, still needs the support of the tools for the LinuxDoc. This “LinuxDoc-Tools” is created for those people. If you need the tools for the LinuxDoc DTD, then you may wish to use this. But remember, the LinuxDoc DTD is not the standard way now even in the Linux world. If you can, try the DocBook DTD. It is the standard, and full-featured way of writing the documentations.

2 Installation

2.1 Where to get the source archive

You can get the source archive of the linuxdoc-tools from:

- `http://www.debian.org/~sano/linuxdoc-tools/`

The name of the archive may be `linuxdoc-tools_x.y.z.tar.gz` or `linuxdoc-tools_x.y.z-rel.tar.gz` or `linuxdoc-tools_x.y.z.orig.tar.gz`. These have the equivalent contents. You can use anyone.

2.2 What LinuxDoc-Tools Needs

LinuxDoc-Tools depends on the usage of sgml parser from Jade or OpenJade (nsgmls or onsgmls). You have to install either of them to use this.

The source archive of the linuxdoc-tools contains the tools and data that you need to write SGML documents and convert them to groff, LaTeX, PostScript, HTML, GNU info, LyX, and RTF. In addition to this package, you will need some additional tools for generating formatted output.

1. `groff`. You *need* version 1.08 or greater. You can get this from `ftp://prep.ai.mit.edu/pub/gnu` . There is a Linux binary version at *ftp://sunsite.unc.edu/pub/Linux/utils/text* as well. You will need `groff` to produce plain text from your SGML documents. `nroff` will *not* work! You can find the version of your `groff` from `groff -v < /dev/null`.

2. TeX and LaTeX. This is available more or less everywhere; you should have no problem getting it and installing it (there is a Linux binary distribution on `sunsite.unc.edu`). Of course, you only need

TeX/LaTeX if you want to format your SGML documents with LaTeX. So, installing TeX/LaTeX is optional. If you need PDF output, then you need pdfLaTeX also.

3. `flex`. `lex` will probably not work. You can get flex from *ftp://prep.ai.mit.edu/pub/gnu* .

4. `gawk` and the GNU info tools, for formatting and viewing info files. These are also available on *ftp://prep.ai.mit.edu/pub/gnu* , or on *ftp://sunsite.unc.edu/pub/Linux/utils/text* (for `gawk`) and *ftp://sunsite.unc.edu/pub/Linux/system/Manual-pagers* (for GNU info tools). `awk` will not work.

5. LyX (a quasi-WYSIWYG interface to LaTeX, with SGML layouts), is available on *ftp://ftp.via.ecp.fr* .

2.3 Installing The Software

The steps needed to install and configure the LinuxDoc-Tools are:

1. First, unpack the tar file of the source archive somewhere. This will create the directory `linuxdoc-tools-x.y.z`. It doesn't matter where you unpack this file; just don't move things around within the extracted source tree.

2. Read the `INSTALL` file - it has detailed installation instructions. Follow them. If all went well, you should be ready to use the system immediately once you have done so.

3 Writing Documents With LinuxDoc-Tools

For the most part, writing documents using LinuxDoc-Tools is very simple, and rather like writing HTML. However, there are some caveats to watch out for. In this section we'll give an introduction on writing SGML documents. See the file `example.sgml` for a SGML example document (and tutorial) which you can use as a model when writing your own documents. Here we're just going to discuss the various features of LinuxDoc-Tools, but the source is not very readable as an example. Instead, print out the source (as well as the formatted output) for `example.sgml` so you have a real live case to refer to.

3.1 Basic Concepts

Looking at the source of the example document, you'll notice right off that there are a number of "tags" marked within angle brackets (< and >). A tag simply specifies the beginning or end of an element, where an element is something like a section, a paragraph, a phrase of italicized text, an item in a list, and so on. Using a tag is like using an HTML tag, or a LaTeX command such as `\item` or `\section{...}`.

As a simple example, to produce **this boldfaced text**, you would type

```
As a simple example, to produce <bf>this boldfaced text</bf>, ...
```

in the source. `<bf>` begins the region of bold text, and `</bf>` ends it. Alternately, you can use the abbreviated form

```
As a simple example, to produce <bf/this boldfaced text/, ...
```

which encloses the bold text within slashes. (Of course, you'll need to use the long form if the enclosed text contains slashes, such as the case with Unix filenames).

There are other things to watch out with respect to special characters (that's why you'll notice all of these bizarre-looking ampersand expressions if you look at the source; I'll talk about those shortly).

In some cases, the end-tag for a particular element is optional. For example, to begin a section, you use the <sect> tag, however, the end-tag for the section (which could appear at the end of the section body itself, not just after the name of the section!) is optional and implied when you start another section of the same depth. In general you needn't worry about these details; just follow the model used in the tutorial (example.sgml).

3.2 Special Characters

Obviously, the angle brackets are themselves special characters in the SGML source. There are others to watch out for. For example, let's say that you wanted to type an expression with angle brackets around it, as so: <foo>. In order to get the left angle bracket, you must use the < element, which is a "macro" that expands to the actual left-bracket character. Therefore, in the source, I typed

```
angle brackets around it, as so: <tt>&lt;foo&gt;</tt>.
```

Generally, anything beginning with an ampersand is a special character. For example, there's % to produce %, | to produce |, and so on. For every special character that might otherwise confuse LinuxDoc-Tools if typed by itself, there is an ampersand "entity" to represent it. The most commonly used are:

- Use & for the ampersand (&),
- Use < for a left bracket (<),
- Use > for a right bracket (>),
- Use &etago; for a left bracket with a slash (</)
- Use $ for a dollar sign ($),
- Use # for a hash (#),
- Use % for a percent (%),
- Use ˜ for a tilde (˜),
- Use ‘‘ and ’’ for quotes, or use &dquot; for ".
- Use for a soft hyphen (that is, an indication that this is a good place to break a word for horizontal justification).

Here is a complete list of the entities recognized by 0.1. Note that not all back-ends will be able to make anything useful from every entity – if you see parantheses with nothing between them in the list, it means that the back-end that generated what you're looking at has no replacement for the entity. The "common" ones listed above are pretty reliable.

&half ($\frac{1}{2}$)

vertical 1/2 fraction

&frac12 ($\frac{1}{2}$)

typeset 1/2 fraction

&frac14 ($\frac{1}{4}$)

typeset 1/4 fraction

&frac34 ($\frac{3}{4}$)

typeset 3/4 fraction

&frac18 ($\frac{1}{8}$)

typeset 1/8 fraction

&frac38 ($\frac{3}{8}$)

typeset 3/8 fraction

&frac58 ($\frac{5}{8}$)

typeset 5/8 fraction

&frac78 ($\frac{7}{8}$)

typeset 7/8 fraction

&sup1 (^{1})

superscript 1

&sup2 (^{2})

superscript 2

&sup3 (^{3})

superscript 3

&plus (+)

plus sign

&plusmn (±)

plus-or-minus sign

&lt (<)

less-than sign

&equals (=)

equals sign

&gt (>)

greater-than sign

&divide (÷)
: division sign

&times (×)
: multiplication sign

&curren ({curren})
: currency symbol

&pound (£)
: symbol for "pounds"

&dollar ($)
: dollar sign

&cent ({cent})
: cent sign

&yen ({yen})
: yen sign

&num (#)
: number or hash sign

&percnt (%)
: percent sign

&amp (&)
: ampersand

&ast (*)
: asterisk

&commat (@)
: commercial-at sign

&lsqb ([)
: left square bracket

&bsol (\)
: backslash

&rsqb (])
: right square bracket

&lcub ({)
: left curly brace

&horbar (–)

horizontal bar

&verbar (|)

vertical bar

&rcub (})

right curly brace

&micro (μ)

greek mu (micro prefix)

&ohm (Ω)

greek capital omega (Ohm sign)

&deg ($^\circ$)

small superscript circle sign (degree sign)

&ordm (º)

masculine ordinal

&ordf (ª)

feminine ordinal

&sect (§)

section sign

&para (¶)

paragraph sign

&middot ($\cdot$)

centered dot

&larr ($\leftarrow$)

left arrow

&rarr ($\rightarrow$)

right arrow

&uarr ($\uparrow$)

up arrow

&darr ($\downarrow$)

down arrow

&copy (©)

copyright

&reg (®)

r-in-circle marl

&trade (™)

trademark sign

&brvbar ({brvbar})

broken vertical bar

&not (¬)

logical-negation sign

&sung ({sung})

sung-note sign

&excl (!)

exclamation point

&iexcl (¡)

inverted exclamation point

&quot (")

double quote

&apos (')

apostrophe (single quote)

&lpar (()

left parenthesis

&rpar ())

right parenthesis

&comma (,)

comma

&lowbar (_)

under-bar

&hyphen (-)

hyphen

&period (.)

period

&sol (/)

solidus

&colon (:)
: colon

&semi (;)
: semicolon

&quest (?)
: question mark

&iquest (¿)
: interrobang

&laquo («)
: left guillemot

&raquo (»)
: right guillemot

&lsquo (‘)
: left single quote

&rsquo (’)
: right single quote

&ldquo (“)
: left double quote

&rdquo (”)
: right double quote

** ( )**
: non-breaking space

&shy ()
: soft hyphen

3.3 Verbatim and Code Environments

While we’re on the subject of special characters, we might as well mention the verbatim “environment” used for including literal text in the output (with spaces and indentation preserved, and so on). The `verb` element is used for this; it looks like the following:

```
<verb>
 Some literal text to include as example output.
</verb>
```

The `verb` environment doesn’t allow you to use *everything* within it literally. Specifically, you must do the following within `verb` environments.

- Use `&ero;` to get an ampersand,
- Use `&etago;` to get </,
- Don't use `\end{verbatim}` within a `verb` environment, as this is what LaTeX uses to end the `verbatim` environment. (In the future, it should be possible to hide the underlying text formatter entirely, but the parser doesn't support this feature yet.)

The `code` environment is much just like the `verb` environment, except that horizontal rules are added to the surrounding text, as so:

```
Here is an example code environment.
```

You should use the `tscreen` environment around any `verb` environments, as so:

```
<tscreen><verb>
Here is some example text.
</verb></tscreen>
```

`tscreen` is an environment that simply indents the text and sets the sets the default font to `tt`. This makes examples look much nicer, both in the LaTeX and plain text versions. You can use `tscreen` without `verb`, however, if you use any special characters in your example you'll need to use both of them. `tscreen` does nothing to special characters. See `example.sgml` for examples.

The `quote` environment is like `tscreen`, except that it does not set the default font to `tt`. So, you can use `quote` for non-computer-interaction quotes, as in:

```
<quote>
Here is some text to be indented, as in a quote.
</quote>
```

which will generate:

> Here is some text to be indented, as in a quote.

3.4 Overall Document Structure

Before we get too in-depth with details, we're going to describe the overall structure of an LinuxDoc-Tools document. Look at `example.sgml` for a good example of how a document is set up.

3.4.1 The Preamble

In the document "preamble" you set up things such as the title information and document style:

```
<!doctype linuxdoc system>

<article>

<title>Linux Foo HOWTO
```

```
<author>Norbert Ebersol, <tt/norb@baz.com/
<date>v1.0, 9 March 1994
<abstract>
This document describes how to use the <tt/foo/ tools to frobnicate
bar libraries, using the <tt/xyzzy/ relinker.
</abstract>

<toc>
```

The elements should go more or less in this order. The first line tells the SGML parser to use the linuxdoc DTD. We'll explain that in the later section on 6 (How LinuxDoc-Tools Works); for now just treat it as a bit of necessary magic. The `<article>` tag forces the document to use the "article" document style.

The `title`, `author`, and `date` tags should be obvious; in the `date` tag include the version number and last modification time of the document.

The `abstract` tag sets up the text to be printed at the top of the document, *before* the table of contents. If you're not going to include a table of contents (the `toc` tag), you probably don't need an `abstract`.

3.4.2 Sectioning And Paragraphs

After the preamble, you're ready to dive into the document. The following sectioning commands are available:

- `sect`: For top-level sections (i.e. 1, 2, and so on.)
- `sect1`: For second-level subsections (i.e. 1.1, 1.2, and so on.)
- `sect2`: For third-level subsubsections.
- `sect3`: For fourth-level subsubsubsections.
- `sect4`: For fifth-level subsubsubsubsections.

These are roughly equivalent to their LaTeX counterparts `section`, `subsection`, and so on.

After the `sect` (or `sect1`, `sect2`, etc.) tag comes the name of the section. For example, at the top of this document, after the preamble, comes the tag:

```
<sect>Introduction
```

And at the beginning of this section (Sectioning and paragraphs), there is the tag:

```
<sect2>Sectioning And Paragraphs
```

After the section tag, you begin the body of the section. However, you must start the body with a `<p>` tag, as so:

```
<sect>Introduction
<p>
This is a user's guide to the LinuxDoc-Tools document processing...
```

This is to tell the parser that you're done with the section title and are ready to begin the body. Thereafter, new paragraphs are started with a blank line (just as you would do in TeX). For example,

```
Here is the end of the first paragraph.

And we start a new paragraph here.
```

There is no reason to use <p> tags at the beginning of every paragraph; only at the beginning of the first paragraph after a sectioning command.

3.4.3 Ending The Document

At the end of the document, you must use the tag:

```
</article>
```

to tell the parser that you're done with the `article` element (which embodies the entire document).

3.5 Internal Cross-References

Now we're going to move onto other features of the system. Cross-references are easy. For example, if you want to make a cross-reference to a certain section, you need to label that section as so:

```
<sect1>Introduction<label id="sec-intro">
```

You can then refer to that section somewhere in the text using the expression:

```
See section <ref id="sec-intro" name="Introduction"> for an introduction.
```

This will replace the `ref` tag with the section number labeled as `sec-intro`. The `name` argument to `ref` is necessary for groff and HTML translations. The groff macro set used by LinuxDoc-Tools does not currently support cross-references, and it's often nice to refer to a section by name instead of number.

For example, this section is 3.5 (Cross-References).

Some back-ends may get upset about special characters in reference labels. In particular, latex2e chokes on underscores (though the latex back end used in older versions of this package didn't). Hyphens are safe.

3.6 Web References

There is also a `url` element for Universal Resource Locators, or URLs, used on the World Wide Web. This element should be used to refer to other documents, files available for FTP, and so forth. For example,

```
You can get the Linux HOWTO documents from
<url url="http://sunsite.unc.edu/mdw/HOWTO/"
   name="The Linux HOWTO INDEX">.
```

The `url` argument specifies the actual URL itself. A link to the URL in question will be automatically added to the HTML document. The optional `name` argument specifies the text that should be anchored to the URL (for HTML conversion) or named as the description of the URL (for LaTeX and groff). If no `name` argument is given, the URL itself will be used.

A useful variant of this is `htmlurl`, which suppresses rendering of the URL part in every context except HTML. What this is useful for is things like a person's email addresses; you can write

```
<htmlurl url="mailto:esr@snark.thyrsus.com"
      name="esr@snark.thyrsus.com">
```

and get “esr@snark.thyrsus.com” in text output rather than the duplicative “esr@snark.thyrsus.com <mailto:esr@snark.thyrsus.com>” but still have a proper URL in HTML documents.

3.7 Fonts

Essentially, the same fonts supported by LaTeX are supported by LinuxDoc-Tools. Note, however, that the conversion to plain text (through `groff`) does away with the font information. So, you should use fonts as for the benefit of the conversion to LaTeX, but don't depend on the fonts to get a point across in the plain text version.

In particular, the `tt` tag described above can be used to get constant-width “typewriter” font which should be used for all e-mail addresses, machine names, filenames, and so on. Example:

```
Here is some <tt>typewriter text</tt> to be included in the document.
```

Equivalently:

```
Here is some <tt/typewriter text/ to be included in the document.
```

Remember that you can only use this abbreviated form if the enclosed text doesn't contain slashes.

Other fonts can be achieved with `bf` for **boldface** and `em` for *italics*. Several other fonts are supported as well, but we don't suggest you use them, because we'll be converting these documents to other formats such as HTML which may not support them. Boldface, typewriter, and italics should be all that you need.

3.8 Lists

There are various kinds of supported lists. They are:

- `itemize` for bulleted lists such as this one.
- `enum` for numbered lists.
- `descrip` for “descriptive” lists.

Each item in an `itemize` or `enum` list must be marked with an `item` tag. Items in a `descrip` are marked with `tag`. For example,

```
<itemize>
<item>Here is an item.
<item>Here is a second item.
</itemize>
```

Looks like this:

- Here is an item.

- Here is a second item.

Or, for an `enum`,

```
<enum>
<item>Here is the first item.
<item>Here is the second item.
</enum>
```

You get the idea. Lists can be nested as well; see the example document for details.

A `descrip` list is slightly different, and slightly ugly, but you might want to use it for some situations:

```
<descrip>
<tag/Gnats./ Annoying little bugs that fly into your cooling fan.
<tag/Gnus./ Annoying little bugs that run on your CPU.
</descrip>
```

ends up looking like:

Gnats.

Annoying little bugs that fly into your cooling fan.

Gnus.

Annoying little bugs that run on your CPU.

3.9 Conditionalization

The overall goal of LinuxDoc-tools is to be able to produce from one set of masters output that is semantically equivalent on all back ends. Nevertheless, it is sometimes useful to be able to produce a document in slightly different variants depending on back end and version. LinuxDoc-Tools supports this through the <#if> and <#unless> bracketing tags.

These tags allow you to selectively include and uninclude portions of an SGML master in your output, depending on filter options set by your driver. Each tag may include a set of attribute/value pairs. The most common are "output" and "version" (though you are not restricted to these) so a typical example might look like this:

```
Some <#if output=latex2e version=drlinux>conditional</#if> text.
```

Everything from this <#if> tag to the following </#if> would be considered conditional, and would not be included in the document if either the filter option "output" were set to something that doesn't match "latex2e" or the filter option "version" were set to something that doesn't match "drlinux". The double negative is deliberate; if no "output" or "version" filter options are set, the conditional text will be included.

Filter options are set in one of two ways. Your format driver sets the "output" option to the name of the back end it uses; thus, in particular, "`linuxdoc -B latex`" sets "output=latex2e", Or you may set an attribute-value pair with the "-D" option of your format driver. Thus, if the above tag were part of a file a file named "foo.sgml", then formatting with either

```
% linuxdoc -B latex -D version=drlinux foo.sgml
```

or

```
% linuxdoc -B latex foo.sgml
```

would include the “conditional” part, but neither

```
% linuxdoc -B html -D version=drlinux foo.sgml
```

nor

```
% linuxdoc -B latex -D private=book foo.sgml
```

would do so.

So that you can have conditionals depending on one or more of several values matching, values support a simple alternation syntax using “|”. Thus you could write:

```
Some <#if output="latex2e|html" version=drlinux>conditional</#if> text.
```

and formatting with either “-B latex” or “-B html” will include the “conditional” text (but formatting with, say, “-B txt” will not).

The <#unless> tag is the exact inverse of <#if>; it includes when <#if>; would exclude, and vice-versa.

Note that these tags are implemented by a preprocessor which runs before the SGML parser ever sees the document. Thus they are completely independent of the document structure, are not in the DTD, and usage errors won't be caught by the parser. You can seriously confuse yourself by conditionalizing sections that contain unbalanced bracketing tags.

The preprocessor implementation also means that standalone SGML parsers will choke on LinuxDoc-Tools documents that contain conditionals. However, you can validity-check them with “`linuxdoc -B check`”.

Also note that in order not to mess up the source line numbers in parser error messages, the preprocessor doesn't actually throw away everything when it omits a conditionalized section. It still passes through any newlines. This leads to behavior that may suprise you if you use <if> or <unless> within a <verb> environment, or any other kind of bracket that changes SGML's normal processing of whitespace.

These tags are called “#if” and “#unless” (rather than “if” and “unless”) to remind you that they are implemented by a preprocessor and you need to be a bit careful about how you use them.

3.10 Index generation

To support automated generation of indexes for book publication of SGML masters, LinuxDoc-Tools supports the <idx> and <cdx> tags. These are bracketing tags which cause the text between them to be saved as an index entry, pointing to the page number on which it occurs in the formatted document. They are ignored by all backends except LaTeX, which uses them to build a .ind file suitable for processing by the TeX utility makeindex.

The two tags behave identically, except that <idx> sets the entry in a normal font and <cdx> in a constant-width one.

If you want to add an index entry that shouldn't appear in the text itself, use the <nidx> and <ncdx> tags.

3.11 Controlling justification

In order to get proper justification and filling of paragraphs in typeset output, LinuxDoc-Tools includes the entity. This becomes an optional or 'soft' hyphen in back ends like latex2e for which this is neaningful.

The bracketing tag <file> can be used to surround filenames in running text. It effectively inserts soft hyphens after each slash in the filename.

One of the advantages of using the <url> and <htmlurl> tags is that they do likewise for long URLs.

4 Formatting SGML Documents

Let's say you have the SGML document `foo.sgml`, which you want to format. Here is a general overview of formatting the document for different output. For a complete list of options, consult the man pages.

4.1 Checking SGML Syntax

If you just want to capture your errors from the SGML conversion, use the "`linuxdoc -B check`". For example.

```
% linuxdoc -B check foo.sgml
```

If you see no output from this check run other than the "Processing..." message, that's good. It means there were no errors.

4.2 Creating Plain Text Output

If you want to produce plain text, use the command:

```
% linuxdoc -B txt foo.sgml
```

You can also create groff source for man pages, which can be formatted with `groff -man`. To do this, do the following:

```
% linuxdoc -B txt --man foo.sgml
```

4.3 Creating LaTeX, DVI, PostScript or PDF Output

To create a LaTeX documents from the SGML source file, simply run:

```
% linuxdoc -B latex foo.sgml
```

If you want to produce PostScript output (via `dvips`), use the "`-o`" option:

```
% linuxdoc -B latex --output=ps foo.sgml
```

Or you can produce a DVI file:

```
% linuxdoc -B latex --output=dvi foo.sgml
```

Also, you can produce a PDF file:

```
% linuxdoc -B latex --output=pdf foo.sgml
```

4.4 Creating HTML Output

If you want to produce HTML output, do this:

```
% linuxdoc -B html --imagebuttons foo.sgml
```

This will produce `foo.html`, as well as `foo-1.html`, `foo-2.html`, and so on – one file for each section of the document. Run your WWW browser on `foo.html`, which is the top level file. You must make sure that all of the HTML files generated from your document are all installed in the directory, as they reference each other with local URLs.

The "`-imagebuttons`" option tells html backend driver to use graphic arrows as navigation buttons. The names of these icons are "next.png", "prev.png", and "toc.png", and the LinuxDoc-Tools system supplies appropriate PNGs in its library directory.

If you use "`linuxdoc -B html`" without the "`-img`" flag, HTML documents will by default have the English labels "Previous", "Next", and "Table of Contents" for navigation. If you specify one of the accepted language codes in a "`-language`" option, however, the labels will be given in that language.

4.5 Creating GNU Info Output

If you want to format your file for the GNU info browser, just run the following command:

```
% linuxdoc -B info foo.sgml
```

4.6 Creating LyX Output

For LyX output, use the the command:

```
% linuxdoc -B lyx foo.sgml
```

4.7 Creating RTF Output

If you want to produce RTF output, run the command:

```
% linuxdoc -B rtf foo.sgml
```

This will produce `foo.rtf`, as well as `foo-1.rtf`, `foo-2.rtf`, and so on; one file for each section of the document.

5 Internationalization Support

The ISO 8859-1 (latin1) character set may be used for international characters in plain text, LaTeX, HTML, LyX, and RTF output (GNU info support for ISO 8859-1 may be possible in the future). To use this feature, give the formatting scripts the "`-charset=latin`" flag, for example:

```
% linuxdoc -B txt --charset=latin foo.sgml
```

You also can use ISO 8859-1 characters in the SGML source, they will automatically be translated to the proper escape codes for the corresponding output format.

Currently, EUC-JP (ujis) character set is partially supported. Source SGML file using this character set can be converted in plain text, HTML, and LaTeX. Other output formats are not tested fully.

6 How LinuxDoc-Tools Works

Technically, the tags and conventions we've explored in previous sections of this use's guide are what is called a *markup language* – a way to embed formatting information in a document so that programs can do useful things with it. HTML, Tex, and Unix manual-page macros are well-known examples of markup languages.

6.1 Overview of SGML

LinuxDoc-Tools uses a way of describing markup languages called SGML (Standard Generalized Markup Language). SGML itself doesn't describe a markup language; rather, it's a language for writing specifications for markup languages. The reason SGML is useful is that an SGML markup specification for a language can be used to generate programs that "know" that language with much less effort (and a much lower bugginess rate!) than if they had to be coded by hand.

In SGML jargon, a markup language specification is called a "DTD" (Document Type Definition). A DTD allows you to specify the *structure* of a kind of document; that is, what parts, in what order, make up a document of that kind. Given a DTD, an SGML parser can check a document for correctness. An SGML-parser/DTD combination can also make it easy to write programs that translate that structure into another markup language – and this is exactly how LinuxDoc-Tools actually works.

LinuxDoc-Tools provides a SGML DTD called "linuxdoc" and a set of "replacement files" which convert the linuxdoc documents to groff, LaTeX, HTML, GNU info, LyX, and RTF source. This is why the example document has a magic cookie at the top of it that says "linuxdoc system"; that is how one tells an SGML parser what DTD to use.

Actually, LinuxDoc-Tools provides a couple of closely related DTDs. But the ones other than linuxdoc are still experimental, and you probably do not want to try working with them unless you are an LinuxDoc-Tools guru.

If you are an SGML guru, you may find it interesting to know that the LinuxDoc-Tools DTDs are based heavily on the QWERTZ DTD by Tom Gordon, `thomas.gordon@gmd.de`.

If you are not an SGML guru, you may not know that HTML (the markup language used on the World Wide Web) is itself defined by a DTD.

6.2 How SGML Works

An SGML DTD like linuxdoc specifies the names of "elements" within a document type. An element is just a bit of structure; like a section, a subsection, a paragraph, or even something smaller like *emphasized text*.

Unlike in LaTeX, however, these elements are not in any way intrinsic to SGML itself. The linuxdoc DTD happens to define elements that look a lot like their LaTeX counterparts—you have sections, subsections, verbatim "environments", and so forth. However, using SGML you can define any kind of structure for the document that you like. In a way, SGML is like low-level TeX, while the linuxdoc DTD is like LaTeX.

Don't be confused by this analogy. SGML is *not* a text-formatting system. There is no "SGML formatter" per se. SGML source is *only* converted to other formats for processing. Furthermore, SGML itself is used only to specify the document structure. There are no text-formatting facilities or "macros" intrinsic to SGML itself. All of those things are defined within the DTD. You can't use SGML without a DTD, a DTD defines what SGML does.

6.3 What Happens When LinuxDoc-Tools Processes A Document

Here's how processing a document with LinuxDoc-Tools works. First, you need a DTD, which sets up the structure of the document. A small portion of the normal (linuxdoc) DTD looks like this:

```
<!element article - -
    (titlepag, header?,
     toc?, lof?, lot?, p*, sect*,
     (appendix, sect+)?, biblio?) +(footnote)>
```

This part sets up the overall structure for an "article", which is like a "documentstyle" within LaTeX. The article consists of a titlepage (`titlepag`), an optional header (`header`), an optional table of contents (`toc`), optional lists of figures (`lof`) and tables (`lot`), any number of paragraphs (`p`), any number of top-level sections (`sect`), optional appendices (`appendix`), an optional bibliography (`biblio`) and footnotes (`footnote`).

As you can see, the DTD doesn't say anything about how the document should be formatted or what it should look like. It just defines what parts make up the document. Elsewhere in the DTD the structure of the `titlepag`, `header`, `sect`, and other elements are defined.

You don't need to know anything about the syntax of the DTD in order to write documents. We're just presenting it here so you know what it looks like and what it does. You *do* need to be familiar with the document *structure* that the DTD defines. If not, you might violate the structure when attempting to write a document, and be very confused about the resulting error messages.

The next step is to write a document using the structure defined by the DTD. Again, the linuxdoc DTD makes documents look a lot like LaTeX or HTML – it's very easy to follow. In SGML jargon a single document written using a particular DTD is known as an "instance" of that DTD.

In order to translate the SGML source into another format (such as LaTeX or groff) for processing, the SGML source (the document that you wrote) is *parsed* along with the DTD by the SGML *parser*. LinuxDoc-Tools uses the `onsgmls` parser in OpenJade, or `nsgmls` parser in Jade. The former is the successor of the latter. `sgmls` parser was written by James Clark, `jjc@jclark.com`, who also happens to be the author of `groff`. We're in good hands. The parser (`onsgmls` or `nsgmls`) simply picks through your document and verifies that it follows the structure set forth by the DTD. It also spits out a more explicit form of your document, with all "macros" and elements expanded, which is understood by `sgmlsasp`, the next part of the process.

`sgmlsasp` is responsible for converting the output of `sgmls` to another format (such as LaTeX). It does this using *replacement files*, which describe how to convert elements in the original SGML document into corresponding source in the "target" format (such as LaTeX or groff).

For example, part of the replacement file for LaTeX looks like:

```
<itemize>    +    "\\begin{itemize}   +
</itemize>   +    "\\end{itemize}    +
```

Which says that whenever you begin an `itemize` element in the SGML source, it should be replaced with

```
\begin{itemize}
```

in the LaTeX source. (As I said, elements in the DTD are very similar to their LaTeX counterparts).

So, to convert the SGML to another format, all you have to do is write a new replacement file for that format that gives the appropriate analogies to the SGML elements in that new format. In practice, it's not that simple—for example, if you're trying to convert to a format that isn't structured at all like your DTD, you're going to have trouble. In any case, it's much easier to do than writing individual parsers and translators for many kinds of output formats; SGML provides a generalized system for converting one source to many formats.

Once `sgmlsasp` has completed its work, you have LaTeX source which corresponds to your original SGML document, which you can format using LaTeX as you normally would.

6.4 Further Information

- The QWERTZ User's Guide is available from *ftp://ftp.cs.cornell.edu/pub/mdw/SGML* . QWERTZ (and hence, LinuxDoc-Tools) supports many features such as mathematical formulae, tables, figures, and so forth. If you'd like to write general documentation in SGML, I suggest using the original QWERTZ DTD instead of the hacked-up linuxdoc DTD, which I've modified for use particularly by the Linux HOWTOs and other such documentation.
- Tom Gordon's original QWERTZ tools can be found at *ftp://ftp.gmd.de/GMD/sgml* .
- More information on SGML can be found at the following WWW pages:
  1. *SGML and the Web* `http://www.w3.org/hypertext/WWW/MarkUp/SGML/`
  2. *SGML Web Page* `http://www.sil.org/sgml/sgml.html`
  3. *Yahoo's SGML Page* `http://www.yahoo.com/Computers_and_Internet/Software/Data_Formats/SGML`
- James Clark's `sgmls` parser, and it's successor `nsgmls` and other tools can be found at *ftp://ftp.jclark.com* and at *James Clark's WWW Page* `http://www.jclark.com` .
- The emacs psgml package can be found at *ftp://ftp.lysator.liu.se/pub/sgml* . This package provides a lot of SGML functionality.
- More information on `LyX` can be found at the *LyX WWW Page* `http://wsiserv.informatik.uni-tuebingen.de/~ettrich/` . `LyX` is a high-level word processor frontend to LaTeX. Quasi-WYSIWYG interface, many LaTeX styles and layouts automatically generated. Speeds up learning LaTeX and makes complicated layouts easy and intuitive.