Let’s make manuals more useful!

Ingo Schwarze

OpenBSD

ABSTRACT

This paper summarizes the lessons learnt from the first six years of mandoc(1) development (2008-2014): what to keep in mind when reading and writing documentation and setting up documentation systems for operating systems and portable software. It covers the content of the tutorial of the same name given on September 26, 2014 during EuroBSDCon in Sofia, in a form more suitable to self-instruction than a set of presentation slides. It includes some additional material extending the scope that wouldn’t fit into the three hours of the tutorial, in particular all relevant content of my two BSDCan presentations given in Ottawa in May 2011 and May 2014.

Version: September 10, 2014

0. Introduction, motivation, and history

0.1 Quality of software and documentation

0.1.1 Quality of software

Obviously, software needs to be correct, robust and secure, or you wouldn't use it for fear of getting incorrect results, being out of service when you need it most, or losing or leaking your data.

Software also needs to be well-designed for usability, or you cannot use it because learning how to use it or actually using it would take too much time and effort.

It directly follows that documentation is a critical, integral part of any software. Without documentation, it is impossible to judge correctness of a piece of software because only the documenation can specify what the software is supposed to do. Without documentation, usability of the software is very bad; it cannot be used at all without experimentation, and even if you figure out the part of the functionality you need, there is no way of making you sure you use it correctly, you use it in a way reliably and securely producing correct results. The only way to be sure would be to read and understand the complete source code, which is impractical in almost all cases, even for a professional software developer.

0.1.2 Quality of documentation

When asked for the quality of a given piece of software, most people think of the quality of the code. Is it well-structured? Is it simple? Was it designed and written with secure coding practices in mind? Is it reliable and robust? Is it efficient?

However, documentation comes with a bunch of additional quality criteria of its own. Obviously, the documentation needs to be correct. A factual error in the documentation is often almost indistinguishable from a bug in the code: In both cases, the software does not do what you expect. Documentation needs to be complete. Features missing from the documentation are not much better than unimplemented or buggy features for the reasons mentioned above. Documentation needs to be concise. The worth of even a brilliantly designed interface is greatly dimnished if you have to toil hours on end trying to make head or tail of the documentation before you can start using the software.

What is often overlooked is that documentation needs to be easy to find and easily accessible.[1] It's a waste of time if you need a web search engine or even locate(1) on the local machine before you find it; and even if you do find it, you may wonder whether it corresponds to the particular version of the software you happen to be using. Unless there is one single standard place for all documentation, you may even miss the existence of some of it. Some documents may require a web browser, some a text processor, some an info(1) program, and if all use different formatting conventions and different controls for navigating it, distraction and misunderstandings will abound.

Software documentation is not just plain text. It embeds and explains syntax elements used by various programming languages and by thousands of utility user interfaces. To ease understanding, such syntax elements need to be marked up, resulting both in special formatting when the manual is displayed and in enabling searches for words in specific syntactic roles.

Finally, documentation needs to be easy to write. There aren't legions of bored technical writers around waiting for new free software being written, waiting to pick it up and document it. And even if they were, they would have a hard time doing the work. Essentially, documenting a piece of software requires reading and understanding the complete code, so the author of the software is about the only person able to adequately and efficiently do the job. Unfortunately, few software developers enjoy writing documentation, most prefer writing code. So in practice, if writing the documentation is difficult or tedious, it will end up being done poorly or not at all.

1. http://www.openbsd.org/papers/bsdcan11-mandoc-openbsd.html
   Almost all content from that presentation is included in the present paper, at some places in updated or corrected form. It is subsequently cited in the following way: BSDCan 2011 p. 2

0.1.3 Case study: OpenBSD

Among the OpenBSD[2] developers, all the points made above are part of a general consensus. Consequently, that project can be used as an example of one way to deal with these points, and to illustrate some of the problems that arise.

Even though the relevance of this paper is by no means limited to OpenBSD, that system will repeatedly be used for illustration purposes throughout this paper, usually in subsections entitled "Case study: OpenBSD". This approach is particularly instructive because almost all of the pioneering work related to the mandoc(1) project[3] was done in the context of the OpenBSD system, paving the way for other systems.

In OpenBSD, all new reference documentation is written in one single format, the mdoc(7) language,[4] and put into one single place, the system manual pages accessible via the man(1) viewer and the apropos(1) search tool. No user-visible code addition, change, or deletion can be committed without updating all documentation affected by the change at the same time. This is easy to handle for developers since all documentation is organized strictly in reference-manual manner. Every piece of documentations closely accompagnies the utility, function, device, or file format it documents. Consequently, all documentation is always complete and up-to-date, even in OpenBSD-current snapshots.

There are some limitations to this system, though. The most obvious one arises when including software having lower documentation standards. The SQLite documentation[5] is a recent and blatant example of such problems. While the documentation provided by SQLite is accurate, complete and concise, fulfilling some quite important requirements, it lacks versioning, ease of access, and syntax markup. It is not included in the software distribution but only made available on the web, so the version you can get access to — if you have Internet access, a browser, and figure out where it is — almost never corresponds to the version of the software you run. On top of that, it critically relies on GIF images[6] that cannot be displayed on a terminal, but require special software. Consequently, OpenBSD contains the SQLite3 library without any documentation. This is an example of an unsolved problem. Automatic or semi-automatic conversion of the documentation to a better format might be possible, but would require very substantial work. The nginx.conf(5) manual[7] is an example where a semi-automatic conversion of web content[8] to an actual manual page was recently performed, and it did indeed require considerable effort.

Another issue with the approach "all documentation is reference manuals" is how to answer questions related to the choice of the right tools, like "does OpenBSD include web servers, and if so, which one is adequate for my purposes?" Users cannot be expected to search for the answer of such a question in the httpd(8) manual, and even if they would, it would be the wrong place for explaining that OpenBSD-specific versions of nginx(8) and Apache 1.3 exist in ports, how these were enhanced with respect to upstream versions, how that might influence the choice of tools, and why you shouldn't use Apache 2 unless you really need it. For questions of this type, OpenBSD has a second place for documentation, the FAQ.[9] There are plans to include it as a special section into the manual, but this has not been done yet. Right now, it's only available on the web, causing the usual versioning problems: It is only updated for each release and does not cover OpenBSD-current. Also, it does not show up when searching the manuals with apropos(1).

0.2 Manuals from the user's perspective

Considering manual pages, for most users, only one single tool will come to mind: man(1), the manual viewer. Maybe surprisingly, until this summer, man(1) was not part of the mandoc toolbox, even though the related search tool, apropos(1), which is also known as **man −k**, was.[10] Very recently, man(1) has finally been integrated.

2. http://www.openbsd.org/
3. http://mdocml.bsd.lv/
4. http://mdocml.bsd.lv/man/mdoc.7.html
5. http://sqlite.org/docs.html
6. http://sqlite.org/images/syntax/select-stmt.gif
7. http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-5.5/man5/nginx.conf.5
8. http://nginx.org/en/docs/dirindex.html
9. http://www.openbsd.org/faq/

Now we have one single tool with a unified, simple user interface, performing the following steps in sequence:

1. Find one or more manuals in the file system. The traditional tools for this purpose are man(1) for retrieval by name and apropos(1) for searches.
2. Transparently call a formatter on them, where required. The traditional formatting commands are mandoc(1) and nroff(1).
3. Display the formatted text, typically in a pager. Again, the traditional tool for this purpose is man(1).

During the last three years, major steps forward for the mandoc toolbox were, from the user perspective:

- Output modes -Tuft8 and -Tlocale are available since May 20, 2011.
- Semantic searching is ready for production since April 14, 2014.
- The new man.cgi(8) featuring semantic searches is online on www.openbsd.org since July 12, 2014.
- A man(1) implementation providing the unified user interface is available in OpenBSD-current since August 26, 2014. However, OpenBSD still installs the traditional man(1) implementation by default.
- The full user interface including mandoc(1) functionality is reachable via the man(1) command name since August 30, 2014.

0.3 Manuals from the author's perspective

Of course, one concern of manual authors will be which languages and tools they have to use to write their manuals.

Nowadays, we recommend using one simple, versatile language for all software documentation: mdoc(7). It is based on the roff(7) language, the roots of which extend well beyond the UNIX era, back to 1964. The mdoc(7) language is the successor of the man(7) language that was first used to format the AT&T Version 7 UNIX manuals in 1979. It was designed for 4.4BSD by the Berkeley Computer Systems Research Group in 1990. It is now supported by both mandoc and groff.[11] Groff development started in 1989, mandoc in 2008.

During the last two years, the main step forward in the mandoc toolbox from the author's perspective was the implementation of the mdoc(7) to man(7) converter. More details on that follow later in this paper.

0.4 Origins of the roff(7) language syntax

Considering the history of the roff language, we can start by celebrating an anniversary. This year, 2014, roff is looking back on exactly half a century of active development.[12]

When preparing his thesis at the MIT, professor Jerome H. Saltzer in 1964 wrote the RUNOFF utility which became later known as roff in the UNIX world.[13] His original implementation used the Michigan Algorithm Decoder (MAD, 1959) programming language for the Compatible Time-Sharing System (CTSS, 1961) operating system running on the MIT's IBM 7094 mainframe computer. He took inspiration from the Memo, Modify, and Ditto tools written by Lowry, Corbato, and Steinberg the year before.

He already used the fundamental concept of text and macro lines. Macro lines contain a period ('.'), a macro name, and optional arguments. They control formatting — and, in modern macro languages, they are also used to specify semantic markup. All other lines are text lines.

---

10. http://www.openbsd.org/papers/bsdcan14-mandoc.pdf
    Almost all content from that presentation is included in the present paper, at some places in updated or corrected form. It is subsequently cited in the following way: BSDCan 2014 p. 2
11. http://www.gnu.org/software/groff/
12. BSDCan 2014 p. 3
13. http://manpages.bsd.lv/history.html

The following requests he introduced in 1964 are still in use today:

ad	select text adjustment mode
br	break the output line
ce	center some lines of text
fi	enable text filling mode
in	indent some lines of text
ll	set the output line length
nf	disable text filling mode
sp	insert vertical spacing

0.5 Advantages of the roff macro syntax

- It can easily be hand-edited with minimal typing overhead.[14]
- It looks unobtrusive and does not muddle the actual text.
- It harmonizes very well with diff(1).
- It allows high quality output in multiple output formats, in particular for terminal output and typesetting.
- It works with simple, fast, portable, readily available tools.
- It does not need any heavyweight or cumbersome toolchains, in particular, it does not require XML.

0.6 Origin of the basic manual structure

Ken Thompson and Dennis M. Ritchie already used roff when preparing the AT&T Version 1 UNIX manual at the Bell Labs in 1971.[15] Dennis M. Ritchie and Joseph F. Ossanna wrote this version of roff for UNIX in DEC PDP-11 assembler.

The format of the manuals was inspired by the CTSS manuals.

The following section headers have been in use since Version 1 UNIX: NAME, SYNOPSIS, DESCRIPTION, FILES, SEE ALSO, DIAGNOSTICS, BUGS.

Until AT&T Version 3 UNIX, the manuals were formatted in pure roff, without using any roff macro set. The only roff source file included was the file *man0/aa*, disabling adjustment with **na**, clearing the hyphenation character with **hc**, setting up a default indentation with **in**, and defining a default page footer line with **fo**. The latter no longer exists in modern roff and would be implemented in terms of **wh** (page position trap) and **tl** (three-part header) nowadays. The only roff requests used abundantly in the Version 3 UNIX manuals were **sp** (insert vertical space) to mark paragraph breaks and **ti** (temporary indent for the next output line) to set section headers without indentation. A considerable list of various macros still in use today occurred occasionally, for example

bp	break output page
br	break output line
ce	center next line
fi	enable fill mode
in	change permanent indent
nf	disable fill mode
nx	include input file
ta	set tab stop positions
tc	set tab fill character
tr	translate character
ul	underline

Precursors to man(7) and mdoc(7) macros occurred in Version 4 to Version 6 UNIX (1973-1975). For example, the

14. BSDCan 2014 p. 5
15. BSDCan 2014 p. 6, http://minnie.tuhs.org/cgi-bin/utree.pl?file=V1/man/manintro.txt

files *man0/naa* for terminal output and *man0/taa* for typesetter output defined macros for the following purposes:

th	page title, now TH/Dt
sh	section header, now SH/Sh
bd	bold text, now B/Sy
it	italic text, now I/Em

The man(7) language first appeared in Version 7 AT&T UNIX (1979).

0.7 Origin of semantic markup in manuals

The mdoc(7) semantic markup macro language was designed to format the manuals of the 4.4BSD release. The translation of the manuals from man(7) to mdoc(7) was performed by Cynthia Livingston of USENIX.[16] The first few of these translated manuals appeared in 4.3BSD-Reno in 1990. The formatter used for this version was Brian Kernighan's device independent troff, written in K&R C, running on BSD UNIX on DEC VAX.

The advantages of the mdoc(7) language are:

- Considerable expressive power for semantic markup, while man(7) is a presentation level language only.
- It works in practice as a standalone language, while man(7) regularly requires resorting to low-level roff features.
- Consequently, mdoc(7) shows a more more uniform appearance and is easier to read and write than man(7).
- Portability is no longer an issue: for legacy systems still not having mdoc(7), mandoc(1) can be used to convert to man(7).
- The mdoc(7) languages supports semantic searching.

0.8 Classic documentation formats (summary)

The roff(7) input syntax, the mdoc(7) semantic markup, and the man(1) presentation format have proven timeless by their simplicity and efficiency.[17] Nobody has come up with a better basic concept yet, even though many have tried, and regarding the formats, there is indeed little one could wish.

Consequently, modern tools are needed for all this.

0.9 Advantages of mandoc

- Functional — all in one binary:[18]
  - Searching by filename, page name, word, substring, regular expression, semantic keys
  - mdoc(7), man(7), tbl(7) and some eqn(7) and roff(7) input
  - ASCII, UTF-8, HTML, XHTML, PostScript, PDF output
  - mdoc(7) to man(7) conversion
  - includes mandoc(1), man(1), apropos(1), whatis(1), and makewhatis(8)
- Free — ISC/BSD-licensed, no GPL code.
- Lightweight — ANSI C, POSIX, no C++ code.
- Portable — includes *compat_*.c* files for missing functions on older systems.
- Small — source tarball (uncompressed) is 8% of groff, executable binary 50%.
- Fast — for mdoc(7), typically 5 times faster than groff, typically about a hundred times faster than an AsciiDoc/DocBook toolchain.

16. BSDCan 2014 p. 7
17. BSDCan 2014 p. 9
18. BSDCan 2014 p. 10

1. Using the mdoc(7) formatting language

1.1 Getting started with mdoc

This section describes the most important things a beginner needs to know about the mdoc(7) language in order to write a new manual page from scratch.

An mdoc(7) source file consists of *macro lines* starting with a dot (‘.’) in the first column, mostly specifying document structure and providing semantic annotations, and *text lines* starting with any other character. Macros take any number of *arguments*, separated by blank characters (‘ ’). If an argument contains a blank character, enclose the whole argument in double quotes (‘"’).

Every manual page starts with a *prologue* exclusively containing macro lines and never any text lines. After that, some *sections* will follow, each starting with an **Sh** macro. Sections have conventional names and follow a strict conventional order. Avoid custom section names, except when splitting the DESCRIPTION in very long manuals.

1.1.1 The mdoc prologue

Always use the following macros in the following order:

Dd	Document date. Use the format *Month day, year* with a full English month name, a one- or two-digit day number, and a four-digit year.
Dt	Document title and section number. The first argument is the name of the manual page that will be passed to the man(1) command, but converted to ALL CAPS. The second argument is the section number as a single digit, see the man(1) or mdoc(7) manuals for lists of section numbers.
Os	Optional operating system name. Just leave it blank.
Sh	Section header. The argument of the first section header macro must be the exact string **NAME**.
Nm	Page name. The same name you provided for **Dt**, but now using its normal lower-case or upper-case spelling, whatever is appropriate.
Nd	One-line description. No quoting is needed.

As a beginner, it may help to remember the number *six*: If your prologue has six lines, it’s probably complete.

Here is a typical example:

```
.Dd July 16, 2013
.Dt CAT 1
.Os
.Sh NAME
.Nm cat
.Nd concatenate and print files
```

1.1.2 The SYNOPSIS section

In manuals documenting utilities (sections 1 and 8) and library functions (sections 2 and 3), the next section always starts with **.Sh SYNOPSIS**. This section only documents syntax, not semantics. It never contains any free text. Do not worry about formatting. Just specify the syntax, formatting will be done automatically.

For utilities, you almost always need the following macros:

Nm	Utility name. Usually the same you used in the NAME section.
Op	Optional syntax element.
Fl	Command line options (flags).
Ar	Command line arguments.

A typical example looks like this:

```
.Sh SYNOPSIS
.Nm cat
.Op Fl benstuv
.Op Ar
```

The formatted output is:

```
SYNOPSIS
     cat [-benstuv] [file ...]
```

Most macros can take other macros as arguments. In that case, the *called* macros don’t have a dot, like **Fl** and **Ar** in the example above. The **Op** macro is an example of an *enclosure* macro, having a *scope* that can contain macros and text. For **Op**, the scope extends to the end of the input line.

For library functions, you almost always need the following macros:

In	Include file.
Ft	Function type.
Fo	Begin (open) function declaration.
Fa	Function argument.
Fc	End (close) function declaration.

A typical example looks like this:

```
.Sh SYNOPSIS
.In unistd.h
.Ft ssize_t
.Fo read
.Fa "int d"
.Fa "void *buf"
.Fa "size_t nbytes"
.Fc
```

The formatted output is:

```
SYNOPSIS
     #include <unistd.h>

     ssize_t
     read(int d, void *buf, size_t nbytes);
```

The **Fo** macro is a *block macro* starting a scope that requires *explicit* closure by the **Fc** companion macro.

1.1.3 The description section

Next, every manual page has a section starting with **.Sh DESCRIPTION**. Start it by explaining the purpose of the topic, followed by a concise description of the syntax and semantics of all features, except those to be described in the following sections coming after the DESCRIPTION itself: RETURN VALUES (of library functions), ENVIRONMENT, FILES, EXIT STATUS (of utilities), EXAMPLES, DIAGNOSTICS (of utilities), ERRORS (of library functions). Finally, you can add some sections containing concluding material: SEE ALSO, STANDARDS, HISTORY, AUTHORS, CAVEATS, BUGS.

1.2 Resources

The most important resource to use is the mdoc(7) manual page. In particular, when wondering which macro to use for markup, first look at the MACRO OVERVIEW section for candidates, then at the description of the specific macro you consider to choose in the MACRO REFERENCE section. The mdoc(7) manual also contains some more details regarding the various manual sections.

To see examples of good usage, look at existing manuals in the OpenBSD base tree. This is particularly helpful when wondering about customary choices of macro arguments. For example, you might find the customary form of an option list in the DESCRIPTION section in this way, which is:

```
    The options are as follows:
    .Bl -tag -width Ds
    .It Fl a
```

Run **mandoc -Tlint** on what you have written. It catches most syntax errors and provides some stylistic hints regarding syntax.

If ambiguities remain even after studying the mdoc(7) manual, try looking at the groff_mdoc(7) manual contained in the textproc/groff port. In some cases, it may contain additional hints.

If you feel information is missing from the mdoc(7) manual or think something could be made clearer, write to the mandoc discussion list.[19]

Kristaps has written a full tutorial, "Practical UNIX Manuals".[20]

1.3 Block nesting

The mdoc(7) language does not only provide formatting cues and semantic markup to manual pages, but it also provides a structure to the document.[21] Sections, subsections, displays, lists, list items, and quoting enclosures are blocks that can nest within each other, and that can contain text and markup elements, which in turn can contain text. So while a man(7) document merely told you "this string is to be set in an italic font", an mdoc(7) document might tell you something like "this is a function argument in a function prototype in the SYNOPSIS section" or "this is an option character marked as optional in a list item of a tagged list in the DESCRIPTION section."

Here is an example of nested blocks:

```
    $ mandoc chgrp.1
    SYNOPSIS
         chgrp [-fh] [-R [-H | -L | -P]] group file ...
```

The corresponding mdoc(7) source code reads as follows:

```
    $ less chgrp.1
    [...]
    .Sh SYNOPSIS
    .Nm chgrp
    .Op Fl fh
    .Oo
    .Fl R
    .Op Fl H | L | P
    .Oc
    .Ar group
    .Ar
    [...]
```

Mandoc represents this by this syntax tree:

```
    $ mandoc -Ttree chgrp.1   # much simplified
        Sh (block) SYNOPSIS
            Nm (block) chgrp
                Op (block)
                    Fl (elem) fh
                Oo (block)
                    Fl (elem) R
                    Op (block)
                        Fl (elem) H
                        (text) |
```

19. mailto:discuss@mdocml.bsd.lv
20. http://manpages.bsd.lv/
21. BSDCan 2011 p. 7

```
                Fl (elem) L
                (text) |
                Fl (elem) P
        Ar (elem) group
        Ar (elem) file ...
```

By contrast, the traditional roff design knew no block structure. On the lower level of that design, *roff requests* provided physical formatting, registers to store data, and macro expansion facilities. On the higher level of that design, the *mdoc macros* called physical formatting requests, set registers to keep state, but did not result in any kind of a syntax tree in any stage of their processing. So even though the mdoc(7) language itself always contained structural information, before the advent of mandoc(1), that information could never actually be used for anything but was discarded in the very first step of the formatting process.

1.4 Badly nested blocks

Unfortunately, and in contrast to many other languages featuring nested blocks, in particular XML, the mdoc(7) language does not only support the nicely nested blocks shown in the previous section, but it also allows badly nested blocks, that is, blocks that overlap without any of them being completely contained in the other.[22]

The simplest case of badly nested blocks can be constructed with just two blocks, if both blocks first open and then close *in the same order*. In that case, text following the opening of the first block is only contained in the first block, text following the opening of the second block is contained in both, and text following the closing of the first block is only contained in the second one.

Here is an example with *explicit* blocks, that is, blocks being closed with an explicit block end macro:

```
.Ao ao
.Bo bo
.No ac Ac
.No bc Bc
```

This example formats as:

```
<ao [bo ac> bc]
```

The same result can be constructed with *implicit* blocks, that is, blocks always extending to the end of their input line and automatically closing their scope at the end of the line. Here is the variant where an explicit block breaks an implicit one, that is, the end macro of the explicit block occurs while the scope of the later implicit block is still open:

```
.Ao ao
.Bq bq ac Ac eol
```

This example formats as:

```
<ao [bq ac> eol]
```

Conversely, an implicit block can break an explicit one, that is, the line containing the implicit block might end before the end macro of the later explicit block arrives:

```
.Aq aq Bo bo eol
.No bc Bc
```

This example formats as:

```
<aq [bo eol> bc]
```

Actually, this is the most important case in practice, and we shall return to it below.

Note that implicit macros cannot break each other. If they are on different input lines, their scopes do not intersect at all. If they are on the same input line, they both end at the same point, at the end of the line, so the earlier one completely contains the later one.

22. BSDCan 2011 p. 8

1.4.1 The scope extension macro

There is one enclosure macro that, in contrast to all other enclosure macros, generates no output whatsoever, neither at the beginning nor at the end of its scope: **Xo**/**Xc**. Its sole purpose is to have its scope broken by an implicit macro, effectively extending the scope of the implicit macro onto the following lines. The only case where this is used in practice is to extend the head scope of the list item macro, .It. Here is a practicle example:

```
$ less find.1
[...]
.It Xo
.Ic -exec Ar utility
.Op argument ...
.No ;
.Xc
```

This example formats as:

```
-exec utility [argument ...] ;
```

For modern roff implementations including groff and mandoc, this can equivalently be written as:

```
.It Ic -exec Ar utility Oo argument ... Oc No ;
```

The original motivation for introducing the **Xo** macro here was that historic roff implementations only supported a limited number of macro arguments on a macro line, and the above one-line version would have violated that limit, causing the last one or two arguments to be lost.

When working on mandoc(1) in 2010, our first thought was: deprecate this abomination, tell manual authors to use the the one-line version, the historical argument limit is no longer relevant, so we get a clean definition of the language. However, large numbers of manuals in various operating systems and in countless portable software packages use **It Xo**, and there is no way to find and change them all, everywhere, or even to change the habits of people writing new manuals.

So somewhat reluctantly at first, we implemented support for badly nested blocks in mandoc(1) in the following way, using the first example cited above for the explanation: The Ao block contains the *whole* Bo block, and the Bo block contains an Ao body-end element in the middle, indicating where Ao formatting is supposed to end.

1.4.2 Case study: OpenBSD

Related timeline:
2010 Feb 23 remove .Oo .Xo .Oc .Xc mis-nesting from manuals (questionable)
2010 Feb 26 support .It Xo (good); all mdoc(7) manuals build now
2010 Jun 29 support badly nested blocks in general (even better)

2. Manual pages for portable software

2.1 Choosing the language

Consider portable software packages like sudo(8), OpenSSH, OpenSMTPd, and so on.[23] Which markup language should be chosen for the manual pages? At first, one might think that choosing mdoc(7) would cause issues for some legacy systems that still don't have mdoc(7) after it has been freely available for more than 20 years (hello, Solaris). However, using man(7) makes maintenance much harder and gives up semantic markup, which would be a very bad idea indeed.

The mandoc toolbox provides a good way out of this dilemma. Write the manual pages using the mdoc(7) language. Use **mandoc −Tman** to convert them to man(7) format — that converter is fully operational since November 19, 2012 and constantly maintained, so you need not fear that it might go away. Include both the mdoc(7) and man(7) versions into distribution tarballs. Let **./configure** figure out what the target system supports and install the best supported version.

2.2 Case study: the sudo(8) manuals

2.2.1 Build system for the distribution tarball

Among the maintainer targets, the Makefile contains targets similar to the following, shown here in simplified form:[24]

```
sudo.man: sudo.mdoc
        mandoc -Tman sudo.mdoc > sudo.man

sudo.cat: sudo.mdoc
        mandoc sudo.mdoc > sudo.cat
```

2.2.2 Installation system

- If **./configure** finds mandoc(1), it installs the **.mdoc* pages.
- If **./configure** does not find nroff(1), it installs the **.cat* pages.
- If **./configure** successfully tests **nroff −mdoc**, it installs the **.mdoc* pages.
- Otherwise, it installs the **.man* pages.
- To override this autodetection logic, it provides options **−-with-mdoc** and **−-with-man**.

2.3 Implementation of the mdoc to man converter

The converter first runs the mdoc(7) parser, constructing exactly the same abstract syntax tree in memory as when running **−Tascii**, **−Thtml**, or **−Tps**.[25]

After that, it runs a dedicated mdoc-to-man output module, structured similarly as the mdoc-to-ASCII output module, but sharing no code. That module consists of only one file, 1600 lines of very straightforward C source code. Its central component is one macro lookup table containing pre- and post-node action functions and pre- and post-node output strings for each mdoc(7) macro type. The module iterates the syntax tree and calls the appropriate action functions for each mdoc(7) node.

An alternative, slightly more flexible approach would have been to first translate the mdoc(7) syntax tree to a man(7) syntax tree, then provide a non-translating man(7) output module. That would have allowed man-to-man code normalization as a by-product. However, the direct approach was simpler and has so far proven sufficient for all practical needs.

23. BSDCan 2014 p. 33
24. BSDCan 2014 p. 34
25. BSDCan 2014 pp. 35

The converter is a typical example of a tool that was technically quite easy to build on top of existing infrastructure, that is on top of the mandoc(3) parser library, but quite useful and powerful in practice.[26] At first, I underestimated the importance of this tool, so development only proceeded haltingly, nearly exclusively at hackathons:

- I started development on September 17, 2011 (s2k11, Ljubljana).
- The bulk of the work was done around July 10, 2012 (g2k12, Budapest).
- It is ready for production since November 19, 2012 (c2k12, Coimbra).

26. BSDCan 2014 pp. 36

3. Quality control for existing manuals

3.1 Why it is critical to get errors and warnings right

A good system of warning and error messages considerably improves the quality of a given piece of software. Without it, even with good documentation, users have a hard time to figure out *what* exactly is wrong as soon as anything goes wrong — and sooner or later, something *will* always go wrong.

There are many pitfalls to avoid when implementing a message system, and all of them cause their specific class of problems for the user:

- A fatal error gets thrown: The manual doesn't format at all, which is very inconvenient. By all means, we want to report errors, but we now try hard to get going and not error out, no matter what broken manual page source code we find.[27]
- An error message is missing or too generic: Users have a hard time to fix their errors.[28]
- A warning message is missing: Users don't even notice their dangerous idioms.
- A few warnings too many, or too prominent: Users get annoyed and switch off all warnings.
- More than one or two knobs: Users don't remember and don't use them.
- Too few and too many can happen all at once! It did with mandoc, and it had too many knobs - at first.

For mandoc(1), improving the message system was an iterative process that took several steps until we arrived at we have today. Major cleanups were done in July 2009, May 2010, August 2010, October 2010, January 2011, March 2011, July 2014, ...

Related timeline:
2009 Jul 12: fewer knobs: remove -Wsyntax -Wcompat
2010 May 13: fewer knobs: remove -fno-ign-chars
2010 May 23: unified error and warning system by kristaps@
2010 Aug 19: simple, consistent user interface for error handling
2010 Oct 24: do not throw fatal errors when there is no need to
2010 Oct 26: downgrade nearly 20 errors to warnings
2011 Jan 16: downgrade yet another bunch of fatal errors
2011 Jan 22: check argument count validation for all in_line() macros

3.2 Goals of quality control for manuals

There are many possible motivations for taking an existing suite of manuals and doing quality checks on them:

Making sure all manuals actually produce output.
    In exceptional cases, manual page files can be so broken that formatters do not produce any output at all, but just show an empty page, abort processing, or even crash, if the broken input triggers bugs in the processors.

Making sure that all intended content is actually shown.
    Some particularly severe markup errors may cause document content to be completely lost during formatting. The most frequent example are probably mistyped macro names. Another example is inadvertently putting text on a preceding macro line instead of starting a new line, exceeding the maximum number of arguments for that particular macro, if it has such a limit.

Catching severe formatting errors.
    Even if all intended text is shown, severe misformatting can make it hard to read.

27. BSDCan 2011 p. 10
28. BSDCan 2011 p. 21

Catching typos and stylistic glitches.
: These can be distracting, sometimes even confusing, when users are reading manuals and trying to understand the content.

Improving portability.
: Some particular constructions work with some formatters but fail outright with other widespread formatters, for example nested displays. Obviously, such constructions should be avoided. Of course, judgement is needed here, portability can easily be overdone. If you were to write manuals in a way compatible with very old, historic formatters, you would end up with mdoc(7) code that would be hard to read and maintain.

Improving robustness.
: Some requests in principle work with all formatters, but are inherently fragile, in particular the **so** (file inclusion) request.

Unifying the style of displayed manuals.
: If the manuals displayed to users follow common conventions with respect to structure, arrangement of content, formatting, and wording, they are easier to understand.

Improving the style of the source code.
: If the mdoc(7) source code of the manual pages follows common conventions, even if these don’t make a difference for what is displayed, editing and maintenance become simpler for authors and developers.

Finding formatter bugs.
: While strictly speaking, this is not a matter of *manual page* quality control, but rather of pager and formatter *utility* quality control, some of the techniques that can be used are so similar that discussing the two together helps a lot. Besides, cases exist where distinguishing formatting bugs, parser bugs, and formatter bugs is non-trivial, or even merely a matter of definition.

Obviously, these goals are very diverse, and the difficulty to do the required checks manually or automatically varies greatly. Various tools are available, most having strengths in one or a few particular areas, but not doing much with respect to other areas. For some goals, there is hardly any automatic support, and manual work is required.

In the following, tools are listed roughly in the order of importance for somebody doing bulk checks on a larger body of manuals. For just checking a single page, you are likely to get away with the simplest of these tools plus some manual checking. On the other hand, for hunting parser bugs, the more elaborate of these tools are certainly needed.

To avoid distraction, the following sections assume that all the source files you want to work on are in a single directory, and that directory contains no other files. In practice, that will sometimes not be the case, and you may need tools like find(1). One useful technique is to prepare a list of files you want to work on and then use commands similar to:

```
mandoc -Tlint -Werror $(cat files.list)
```

3.3 Checking with mandoc -Tlint

The most important tool for any kind of manual page quality control is mandoc(1) itself, in particular its **−W** and **−Tlint** options.

3.3.1 Catching fatal errors

When processing a larger amount of manual pages, the first step is to check whether any of these pages causes a fatal error. When dealing with just a handful of pages, this step can be skipped, the fatal errors will show up among the normal errors below.

To find fatal errors, run a command similar to:

```
mandoc -Tlint -Wfatal *
```

Any fatal errors that show up here must be fixed and should be dealt with first, before looking at anything else. A fatal error means that the user is unable to read the manual at all, because parsing the manual fails outright and no output whatsoever can be generated. Consequently, having a manual containing a fatal error is not much better than having no manual at all.

Very few types of fatal errors exist. Those that may occasionally occur in practice are all related to file inclusion:

- use of the unsafe macro **Bd −file**
- use of the **so** file inclusion request with an absolute path
- use of **so** with a path containing ".."
- **so** pointing to a file that doesn’t exist or can’t be opened

See the mandoc(1) manual in the portable mandoc distribution[29] for details, section "FATAL errors".

3.3.2 Catching errors

The most important step when doing quality control with mandoc(1) is to deal with errors. These are issues that can potentially cause loss of information, severe misformatting, or severe portability problems.

To find errors, run a command similar to:

```
    mandoc -Tlint -Werror *
```

The mandoc(1) utility tries very hard to avoid false positives when reporting errors. So, if it reports any errors, you very probably want to fix them. If you run into anything you did on purpose and want to keep it as it is but mandoc(1) calls it an error, please do report that to the mandoc developers.

There are more types of errors than fatal errors, but the number of error types that occur in practice is still rather small:

- unencoded non-ASCII characters in the input
- unknown or mistyped macro or request names
- blocking issues:
  - opening blocks that are never closed again
  - closing blocks that were never opened
  - items outside lists
  - bad nesting of blocks that don’t support it
- severe issues with macro or request arguments:
  - missing essential arguments
  - invalid arguments that cannot be handled adequately
  - excessive arguments that get completely lost during formatting

Again, see the mandoc(1) manual in the portable mandoc distribution for details, section "ERRORS".

3.3.3 Mandoc warnings

Everything else mandoc(1) considers problematic is classified as a warning.

To see the warnings, run a command similar to:

```
    mandoc -Tlint *
```

In any tree containing many low-quality manuals, this is likely to produce a lot of output. On the other hand, the complete set of 145 OpenBSD system call (section 2) manuals currently causes only 28 warnings grand total. Of these 145 manuals, 141 have no warnings at all.

We do try to avoid bogus warnings, but sometimes we fail. Also, some usage that is usually a bad idea may be justified in exceptional cases, in which case an occasional false positive is the price to pay to avoid frequent false negatives.

So, fixing all warnings is usually a good idea, but don’t do so blindly, there may be exceptions.

29. http://mdocml.bsd.lv/man/mandoc.1.html

Warning types are too numerous to provide an exhaustive list here, but most are of these or similar kinds:

- structural errors and syntax errors that only have local effects and do not cause information loss
- low quality syntax like badly nested blocks or macro usage in contexts expecting plain text
- macros that have no effect or are slightly misplaced
- missing arguments or information, if the effect is only local
- violations of usual structural or formatting conventions
- warnings about robustness and portability
- dubious usage of white space and comments

3.4 Checking with mdoclint

The mdoclint(1) utility by Thomas Klausner has a similar focus as **mandoc −Tlint**. It contains some additional tests, so it's a useful complement. It also tries to avoid false positives, though not quite as strictly as **mandoc −Tlint**. So the same caveat applies: Do not slavishly follow its findings.

The mdoclint(1) utility is available on NetBSD[30] and OpenBSD[31] and ought to be trivial to port to other systems providing the perl(1) programming language.

To run it, simply give the names of the files you want to check on the command line. Options are almost exclusively needed if you want it to shut up about some of its findings.

3.5 Checking with igor

The igor(1) utility[32] by Warren Block has a completely different focus than the two linters: It mostly cares about style and spelling and knows relatively little about syntax.

In contrast to the two linters, igor(1) isn't afraid of false positives. Consequently, it is almost unusable for bulk verifications of large trees of high-quality manuals because any signal will likely be drowned in lots of noise.

For finding candidates of bad style or spelling in smaller sets of manuals, it is quite useful and finds whole classes of issues the two linters are completely unaware of.

It is available as a port in both FreeBSD[33] and OpenBSD[34] and should also be trivial to port to any system having perl(1).

Running it works almost the same way as for mdoclint(1). Simply provide the names of the files to be checked as command line arguments. To suppress unwanted messages, use command line options.

3.6 Groff-mandoc comparisons

Comparing the output of groff(1) and mandoc(1) for the same input file is an important way of finding bugs in the parsers and formatters. In some cases, such differences can also provides hints that code is of limited portability.

Of course, if code formats differently, that doesn't necessarily imply that the code can be improved. While it's sometimes easy to spot parser and formatter bugs in such comparisons, interpreting them to identify markup of questionable quality definitely requires wide experience with the mdoc(7) language.

A very simple but handy shell script to run such comparisons, **gmdiff**, is available from the portable mandoc repository.[35] Simply provide the names of the files to check as command line arguments.

30. http://cvsweb.netbsd.org/bsdweb.cgi/pkgsrc/textproc/mdoclint/
31. http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/regress/usr.bin/mdoclint/
32. http://en.wikipedia.org/wiki/Igor_%28character%29
33. http://svnweb.freebsd.org/ports/head/textproc/igor/
34. http://cvsweb.openbsd.org/cgi-bin/cvsweb/ports/textproc/igor/
35. http://mdocml.bsd.lv/cgi-bin/cvsweb/gmdiff?cvsroot=mdocml

3.7 Consistency checking with makewhatis

While building databases, makewhatis(8) can watch out for the following issues:

- Mismatch of the section number given in the manual page with the directory the page is stored in.[36]
- Mismatch of the architecture name given in the manual page with the directory the page is stored in.
- File name does not appear as a name in the NAME section (since April 4, 2014).

The checks that were already performed by the old makewhatis(8) are preserved:

- Missing NAME section, missing name(s) and/or missing description.
- A name in the name section does not appear as an MLINK in the file system (since April 4, 2014).

Besides, direct inspection of the database has been used to catch markup errors.

With makewhatis(8), more can be done later, all this is just a start.

36. BSDCan 2014 p. 24

4. Searching and displaying manual pages

4.1 Backward compatibility

The mandoc toolbox preserves the existing functionality of traditional search tools:[37]

apropos *keywords ...*
Search case-insensitively for substrings in names and descriptions (working since October 19, 2013).

man −k *arguments*
As before, an alias for: **apropos** *arguments*
But it now also supports new-style arguments, see below.

apropos [**−C** *file*] [**−M** *path*] [**−m** *path*] [**−S** *arch*] [**−s** *section*] *keywords ...*
Traditional options are all supported.

whatis *keywords ...*
Search case-insensitively for complete words in page names only.

makewhatis
Rebuild all configured databases.

makewhatis −d *directory files ...*
Update entries for the given *files* in one database (working since June 3, 2013).

4.2 Markup-sensitive search

In addition to traditional functionality, the mandoc toolbox supports searching in specific semantic contexts identified by macro keys. The following are examples of macro keys that can be searched for, ordered by frequency:[38]

Nm	manual page names
Nd	manual page descriptions
sec	manual section numbers
arch	machine architectures
Xr	cross references
Ar	command argument names
Fa	function argument types and names
Dv	preprocessor constants
Pa	file system paths
Cd	kernel configuration directives
Va	variable names
Ft	function return types
Er	error constants
Ev	environment variables
In	include file names
St	references to standards documents
An	author names
...	and so on ...

Here is an example of a search command and its results:

```
 $ apropos Ev=USER
Mail, mail, mailx(1) – send and receive mail
csh(1) – a shell (command interpreter) with Clike syntax
login(1) – log into the computer
```

37. BSDCan 2014 p. 12
38. BSDCan 2014 p. 13

```
logname(1) – display user’s login name
slogin, ssh(1) – OpenSSH SSH client (remote login program)
su(1) – substitute user identity
[...]
```

4.3 Markup-sensitive search features

Search keys can be OR’ed by simply joining them with commas, using a common search string for all of them:[39]

```
 $ apropos Fa,Ft,Va,Vt=timespec
EV_SET, kevent, kqueue(2) – kernel event notification mechanism
clock_getres, clock_gettime, clock_settime(2) – get/set/calibrate date and time
futimens, futimes, utimensat, utimes(2) – set file access and modification times
nanosleep(2) – high resolution sleep
parse_time(3) – parse and unparse time intervals
poll, ppoll(2) – synchronous I/O multiplexing
pselect, select(2), FD_CLR, FD_ISSET, FD_SET, FD_ZERO(3) – synchronous I/O multiplexing
sem_timedwait, sem_trywait, sem_wait(3) – decrement (lock) a semaphore
tstohz, tvtohz(9) – translate time period to timeout delay
[...]
```

Searching across all macro keys is possible with the special **any** keyword:

```
 $ apropos any=ulimit
ksh, rksh(1) – public domain Korn shell
sh(1) – public domain Bourne shell
getrlimit, setrlimit(2) – control maximum system resource consumption
```

Regular expressions are supported by using the ‘˜’ operator instead of ‘=’ since October 19, 2013:

```
 $ apropos "Nm˜^[gs]et.*gid"
endgrent, getgrent, getgrgid, getgrgid_r, getgrnam, getgrnam_r, setgrent, setgrfile, setgro
getegid, getgid(2) – get group process identification
getpgid, getpgrp(2) – get process group
getresgid, getresuid, setresgid, setresuid(2) – get or set real, effective and saved user o
setegid, seteuid, setgid, setuid(2) – set user and group ID
setpgid, setpgrp(2) – set process group
setregid(2) – set real and effective group IDs
[...]
```

4.4 Complex search queries

By default, multiple search terms are joined with OR, but the **−s** and **−S** options attach to the rest of the search expression with AND:[40]

```
 $ apropos −s 1 tbl Nm=eqn
deroff(1) – remove nroff/troff, eqn, pic and tbl constructs
eqn(1) – format equations for troff or MathML
eqn2graph(1) – convert an EQN equation into a cropped image
neqn(1) – format equations for ascii output
tbl(1) – format tables for troff
```

39. BSDCan 2014 p. 14
40. BSDCan 2014 p. 15

Explicit logical AND and OR are supported:

```
 $ apropos Nd=gigabit −a Cd=sbus
gem(4) – GEM 10/100/Gigabit Ethernet device
ti(4) – Alteon Networks Tigon I and II Gigabit Ethernet device
```

Precedence can be changed with parantheses:

```
 $ apropos −s 1 terminal −a \( At˜[1−6] −o Bx˜ˆ[12] \)
clear, tput(1) – terminal capability interface
lock(1) – reserve a terminal
reset, tset(1) – terminal initialization
script(1) – make typescript of terminal session
stty(1) – set the options for a terminal device interface
tty(1) – return user’s terminal name
```

Complex search queries are working since January 4, 2014.

4.5 Flexible output format

The names, section numbers, and architectures of the search results are always shown because these are needed to access the results with man(1).[41] By default, apropos(1) also shows the one-line descriptions.

With the **−O** option, any other macro key can be shown instead:

```
 $ apropos −O Cd wireless
acx(4) – acx* at pci? # acx* at cardbus?
an(4) – an* at isapnp? # an* at pcmcia? # an* at pci?
ath(4) – ath* at pci? dev ? function ? # ath* at cardbus? dev ? function ? #
athn(4) – athn* at cardbus? # athn* at uhub? port ? # athn* at pci?
atu(4) – atu* at uhub? port ?
atw(4) – atw* at pci? # atw* at cardbus?
```

The **−O** option is available since December 31, 2013.

4.6 Unified user interface including man(1)

The very latest development is that the mandoc toolbox now provides an implementation of man(1) and a unified interface for mandoc(1), man(1), whatis(1), and apropos(1). That is, all the command line options of all these utilities are available in all of them and have the same meaning everywhere, and almost all functionality is available from all command names, even though the default behaviour of the different command names is still different.

4.6.1 How it works

The new unified main program always uses a five-step process:

1. Decide how to interpret the command line arguments.
2. Build a list of manual pages, usually from a database search.
3. Decide which kind of output to provide.
4. Optionally spawn a pager.
5. Loop around the list of manual pages, producing some output for each.

41. BSDCan 2014 p. 16

Some **input options** are available to specify the meaning of the command line arguments:

- **−l** Interpret each command line argument as an (absolute or relative) filename. No database search is done. This is the default when called as **mandoc**.
- * The default when called as **man** without an input option is to interpret each command line argument as a name and require exact matches (as opposed to word, substring, or regular expression matches). The default output mode in this case is to show exactly one manual considered the best match. Currently, there is not (yet?) any option to force this behaviour, so it’s only available when called as **man** for now.
- **−f** Interpret each command line argument as a name and match against complete words (as opposed to substring or regular expression matches). This is the default when called as **whatis**.
- **−k** Support the full apropos(1) search syntax. This is the default when called as **apropos**. The default output mode in these two cases is to display a list of names, section numbers, and description lines of matching manuals.

Some **database selection** options only matter when **−l** is not active:

- **−C** Use the specified *file* instead of the default configuration file.
- **−M** Use the specified *path* instead of the default one. Do not use any configuration file.
- **−m** Use the specified *path* in addition to the default one.
- **−S** Restrict the search to the specified *architecture*.
- **−s** Restrict the search to the specified *section*.

Some **output options** are available to specify which kind of output to provide:

- **−a** Display all matching manual pages, one after the other. This is the default for **−l** input mode.
- **−h** Display only the SYNOPSIS lines of the matching pages. Implies **−a**.
- **−O** When showing a list in **−f** or **−k** mode, display the specified macro key instead of the **Nd** one-line descriptions.
- **−w** Display only the pathnames of the matching manual pages.

Some **parser and formatter options** only take effect when a parser is actually run:

- **−I** Override the default operating system name for the mdoc(7) **Os** macro.
- **−m** Specify the input format. Defaults to **−mandoc**, requesting autodetection.
- **−O** Comma-separated formatter-specific output options.
- **−T** Select the output format. Defaults to **−Tascii**.
- **−W** Specify the minimum message *level* to be reported on the standard error output and to affect the exit status. Defaults to **−Wfatal**.

Finally, the **−c** option can be used to suppress the pager and just copy the formatted manuals to standard output.

4.6.2 How this came about

Four months back at BSDCan 2014 in Ottawa, i presented a slide "possible future directions". This project was *not* listed. I didn’t expect myself that i would do this.

But then, on August 9 this year (less than two months ago now) Paul Onyschuk of Alpine Linux (which is the first Linux distro that integrated mandoc, in July 2010) asked me: “Are there any plans for providing a man(1) command also? This would make mdocml a possible, standalone replacement for the groff and man-db combination (typical in Linux distributions).”

Almost, i returned my standard negative answer, but then i stopped short and realized that almost all the needed code was already there and it cheaply allows doing fancy things without complexity. I had to do the mandoc 1.13.1 and 1.12.4 releases first, so it took two weeks from the idea to the first working implementation...

4.6.3 What a ‘name’ is

This question is critical because the man(1) command is supposed to display manual pages whose names exactly match the command line arguments. The traditional man(1) utility only uses filenames as names in this sense.

When makewhatis(8) runs, it adds names to the *names* table in the mandoc.db(5) databases, noting in the *names.bits* column of that table where the name came from (**Dt**/**TH** header line, **Nm** in the NAME section, **Nm** in the

SYNOPSIS section, file name). It also saves the file names where stuff was found into the *mlinks* table and links both tables together (and to the *keys* table with all the search terms) via the *mpages* table.

For man(1) mode, right now, all types of names are used. That can easily be tuned, and probably it should.

- The name from the **Dt**/**TH** is probably not all that useful for this purpose because it lacks the case information.
- The **Nm** macros from the NAME section should almost certainly be used. That way, all hard links, symbolic links, and .so link files become obsolete. Consequently, the number of files in a typical operating system installation can be reduced by more than three thousands.
- The **Nm** macros from the SYNOPSIS section should probably not be used for this purpose.
- The file names should probably still be used, just in case someone installs manuals the old way and screws up the name sections.

4.6.4 Which issues remain

- The **−i** (interactive) option hasn't been integrated yet, that code is still kept separately in the manpage(1) utility which was never used in any kind of system integration.
- Most man.conf(5) features are not supported. The following types of lines are completely ignored: **_build**, **_default**, **_subdir**, **_suffix**, and *section* lines. The search order 1, 8, 6, 2, 3, 5, 7, 4, 9 is hardcoded. Instead of **_default**, the **_whatdb** lines are used. All subdirectories are hardcoded to `{man,cat}N`.
- When finding a formatted and an unformatted manual of the same name in the same section, the old man(1) shows the one that was less recently changed. The mandoc man(1) currently always prefers the unformatted version, even if it's older than the formatted version.
- The mandoc man(1) may show some additional results, and some of those may be bogus, if NAME sections contain bogus **Nm** macros.
- The mandoc man(1) ignores the MACHINE environment variable, and i'm not planning to add support for it.

4.7 Web interface for manual search and display

The man.cgi(8) program contained in the mandoc toolbox provides the same user interface as on the command line. It has a man(1) and an apropos(1) mode, the latter using the same query syntax.

The main additional feature of the web interface are hyperlinks generated for cross-page (**Xr**) and in-page (**Sx**) cross-references. The code has additional potential because it preserves semantic markup, but that's not used yet for anything except (simple) CSS formatting.

The man.cgi(8) program uses the same directory structure and database format as the command line tools, which makes setup rather easy. Configuration instructions are provided in the man.cgi(8) manual.

Of course, setting up a man.cgi(8) server only makes sense for providers of operating systems and of major software packages having many manuals: Do not run your own server with a copy of the manuals of your favourite system, that would merely add a risk to get outdated and confuse people.

Special thanks goes to *Sébastien Marie* for doing an extensive security audit of the man.cgi(8) code and reporting a considerable number of security-relevant bugs that have all been fixed by now.

Related timeline:
2011 Nov 09: kristaps@ starts development of man.cgi
2012 Mar 23: first mandoc release containing man.cgi (1.12.1)
2012 Jun 08: kristaps@ starts moving the database backend to SQLite
2014 Jul 09: schwarze@ switches over man.cgi to SQLite
2014 Jul 12: new man.cgi running on the openbsd.org website
2014 Aug 10: first mandoc release containing the SQLite-based man.cgi (1.13.1)

4.8 Database implementation

The old *whatis.db* was a plain text file.[42] Now we need a structured database, but a client-server model would be overkill and merely a hindrance. So SQLite[43] was the logical choice.

The database contains four tables. They contain, respectively, one record ...

mpages	... per physical page, containing the description
mlinks	... per file system entry, containing section, architecture, filename
names	... per page name; for all manual page names, not just file names
keys	... per key=value pair

The mlinks table has full support for:

- hard links since December 27, 2013
- symbolic links since April 18, 2014
- redirections using the roff .so request since March 19, 2014 (used by X.org)

4.9 Search algorithm

Each search requires either two or three queries:[44]

1. `SELECT FROM mpages`
   to find the pages to be displayed.

   Conveniently, searching for descriptions is fastest — in the first step, access one single table only, finding the *pageid*. Searching for names is the second in speed — it requires only a simple JOIN to a small table.

2. `SELECT FROM names`
   to find the page names to be displayed.

   This is very fast because it is just a simple SELECT in a small table using the indexed *pageid*.

3. `SELECT FROM keys`
   to find the values to be displayed.

   Only needed when **−O** is given. Otherwise, we already have the description from search step 1. Very fast, too, just another simple SELECT indexed by *pageid*.

4.10 Optimization

As usual, optimization is not a well-defined task in the mathematical sense.[45] We want high speed and small size:

- a small database
- a short database build time
- low apropos memory consumption
- short search times

Of course, these optimization goals conflict.

The gprof(1) profiler was used a lot.

42. BSDCan 2014 p. 17
43. http://www.sqlite.org/
44. BSDCan 2014 p. 18
45. BSDCan 2014 p. 19

4.10.1 Search speed optimization

- Moving the **descriptions** from the keys table directly into the mpages table gained roughly a **factor 4** in speed for searches by description — at no cost, the database shrank, too, due to reduced pageid overhead (April 9, 2014).[46]
- The dedicated **names** table gained roughly a **factor 4** in speed for searches by name — at almost no cost (April 9, 2014).
- Adding an index to the **mlinks** table sped up the second step in the algorithm, name retrieval, by about a factor of 20, which resulted in an overall **30% economy** for simple searches, and more for searches returning many results. The cost was a 10% growth of the database (April 16, 2014).
- Adding an index to the **keys** table sped up the third step in the algorithm, **−O** value retrieval, which is dominant in **−O** searches, resulting roughly in a **factor 4** speedup of such searches. The cost was another 10% grow of the database.
- By providing an SQLITE_CONFIG_PAGECACHE with mmap(3) MAP_ANON, execution time decreased by 20−25% for simple (Nd and/or Nm) queries, 10−20% for non-NAME queries, and even apropos(1) resident memory size decreased by 20% for simple and by 60% for non-NAME queries. Cache size is a compromize to provide nearly optimal speed gain for all queries while limiting additional memory consumption to about 15% (April 11, 2014).

4.10.2 Database build time optimization

This is relevant because mandoc.db(5) is built during regular base system and Xenocara snapshots builds on all architectures, and we don't want to slow down developers during development and testing.[47]

- Quick mode: Abort parsing after the NAME section: factor 2 in speed and factor 4 in size (January 5, 2014). In the following, all speedups refer to quick mode.
- Do not sync to disk after each individual manual page, only sync to disk one single time when all data is ready: 87% (January 6, 2014).
- In quick mode, do not clear user-defined macros clashing with mdoc(7) or man(7) standard macros when parsing .Dd or .TH: 25% (January 6, 2014).
- In quick mode, do not validate and normalize the date format: 18% (January 6, 2014).
- Do not copy predefined strings into the dynamic string table: 10% (January 6, 2014).
- Cache uname(3) result: 3% (January 7, 2014).
- Do not index the keys in the keys table: 12% (and 42% size reduction) (January 18, 2014).
- No primary keys in the mlinks and keys table: 15% (and 3% size) (January 18, 2014).
- Properly handling .so redirections reduced Xenocara database build time by 40% and database size by nearly 50% (March 19, 2014).

46. BSDCan 2014 p. 20
47. BSDCan 2014 p. 21

4.10.3 Database size optimizations

All size reductions refer to quick mode.[48]

- Do not store the descriptions twice, once for searching and once for display, at the expense of somewhat more complicated, but not slower search code: 9% (January 5, 2014).
- Remove the redundant "file" column from the mlinks table: 9% (January 5, 2014).
- Sort macro keys by frequency: 11% (January 18, 2014).
- Always store the arch in lower-case only: 1.5% (January 19, 2014).

4.10.4 Search and database performance summary

With the old, plain text apropos(1), a simple search took about 10 milliseconds on my notebook.[49] With the new, SQLite apropos(1), it is unavoidably slower due to the SQL overhead and because the names are now separated from the descriptions. It now takes about 40 milliseconds. However, the difference is of no practical relevance even on moderately old hardware.

Base system database size grows from 250 kB to 900 kB (quick mode) or about 3800 kB (fully featured mode). That is not a practical problem for any of our architectures. During system builds, database build times are reduced by roughly a factor 3 with respect to the old Perl makewhatis(8).

48. BSDCan 2014 p. 22
49. BSDCan 2014 p. 23

5. Integrating mandoc as a base system documentation formatter

5.1 Preparations

5.1.1 Preliminary note

More than three years ago, when summarizing the status of the groff to mandoc switch at BSDCan 2011, i already said this:[50]

In OpenBSD, we now reached the stage: It just works. To get there, we

- re-implemented a small family of languages.
- turned the whole paradigm of these languages upside down.
- remained compatible where it matters.
- flouted compatibility that would just hinder.

Other systems can now do the same, if they want:

- without much risk.
- without having to fear major surprises.
- getting support from us in case of need — just mail us!

Meanwhile, NetBSD and illumos have switched from groff to mandoc, too, in 2012 and 2014, respectively. So you certainly don't have to fear the switch, nowadays. Just follow the plan laid out below.

5.1.2 Getting started

The first step is to import the mandoc codebase into the base repository of the target operating system, collect some initial experience using it, and enable it to be built and installed by default.

The most important aspect of this phase is to establish the contact to everybody in the operating system's developer community who is interested and might wish to be involved. Get familiar with specific requirements of the system and any specific topics the developers are concerned about. Do not neglect this aspect or it will bite you later.

This phase also provides a great opportunity to find bugs in mandoc, to report them, and to get them fixed, without any of the stress that might arise from already being in production, or even from having made specific plans when to go into production.

5.1.3 Case study: OpenBSD

In OpenBSD, this phase lasted from June to December 2009.[51] Because mandoc has matured since that time, all this ought to be much easier now. In any case, this phase can be shorter. Start the next phase early, it will help you to find bugs and to focus on the most relevant ones.[52]

In OpenBSD, we also used this phase to start a regression suite.[53] At first, naturally, we encountered lots of new bugs and few regressions. Consequently, automatic tests were not a priority. However, we started collecting anyway. As the code matured, the number of tests slowly increased, and the suite slowly became more important. Maintenance was done at intervals to keep the suite neat and tidy as it grew.

Already in 2011, the suite regularly caught issues when doing major merges. It took less than two years for the continuous effort to start to pay off. By now, the suite is an indispensable tool used whenever applying any changes to the mandoc codebase.

50. BSDCan 2011 p. 20
51. BSDCan 2011 p. 5
52. BSDCan 2011 p. 10
53. BSDCan 2011 p. 22

Having that suite available on your system would definitely be helpful. Unfortunately, a portable version of the regression suite does not yet exist. Setting it up would be a worthwhile project. Ideally, the tests should not be copied; they would quickly get outdated and require constant maintenance. Rather, one should try to provide Makefile fragments (or something similar) to help the OpenBSD suite to run on other systems, at least on those having a BSD make(1).

Related OpenBSD timeline:
2008 Nov 22: first commit to mdocml.bsd.lv by kristaps@
2009 Mar 27: first direct commit by schwarze@ to OpenBSD (not mandoc)
2009 Apr 06: mandoc imported into OpenBSD by kristaps@
2009 Apr 15: first help from another OpenBSD developer (miod@)
2009 May 23: schwarze@ first talks to jmc@ about mandoc
2009 May 31: at c2k9 in Edmonton, schwarze@ talks to deraadt@
2009 Jun 09: kristaps@ agrees to work closely together
2009 Jun 14: merge to OpenBSD started by schwarze@
2009 Jun 15: first patches merged back from OpenBSD to bsd.lv
2009 Jun 21: mandoc usable in OpenBSD and in sync with bsd.lv
2009 Jun 23: bugfixing in OpenBSD started by schwarze@
2009 Jul 05: OpenBSD 4.6 release rolled without mandoc
2009 Jul 12: joerg@ sends his first patch from NetBSD
2009 Jul 18: uqs@ sends his first patch from FreeBSD
2009 Oct 27: start src/regress/usr.bin/mandoc
2010 Jan 02: start of systematic integration
2010 Jun 30: major update of the mandoc test suite
2010 Jul 01: enable mandoc regression tests; ok phessler@
2010 Dec 04: major additions to the regression suite
2011 Feb 05: commit many regression tests found in my trees

5.2 Identify fatal issues

Try to build all manuals of the target operating system with mandoc. Ignore all non-fatal issues for now. Report all fatal errors to the mandoc developers. Have them fixed upstream or devise workarounds. We are glad to provide help! Only as a last resort, change the affected manuals in your tree.[54]

5.2.1 Case study: OpenBSD

In OpenBSD, this phase took place in January and February 2010.[55] Since mandoc hardly throws any fatal errors any longer, this phase ought to be much easier nowadays. None of the fatal errors that held us back in OpenBSD in 2010 and that i listed in my 2011 BSDCan talk is still fatal today. The mandoc parser is now able to recover from all of them and keep going, so they could all be downgraded to non-fatal errors and some even to mere warnings.

Related timeline:
2010 Jan 02: first patches to mdoc(7) manuals to fix the build with mandoc
"Fine. Even if mandoc goes nowhere, it has found some bugs. ;)" jmc@
2010 Feb 17: first patch to a man(7) manual in order to fix the build
2010 Feb 20: found first manual bug caused by DocBook
2010 Feb 24: first non-fatal manual fix found by -Tlint
2010 Feb 25: tree now builds with mandoc
2010 Mar 18: OpenBSD 4.7 release rolled without mandoc

54. BSDCan 2011 p. 10
55. BSDCan 2011 p. 6

5.3 Investigate non-fatal errors

Run **mandoc −Tlint −Werror** on all manuals in your system. Prioritize fallout that ruins content or formatting. Distinguish mandoc bugs from markup bugs. Report mandoc bugs upstream.

The most serious issues showing up in this phase typically do so in man(7) manuals, not in mdoc(7) manuals. This is not a coincidence. While the mdoc(7) language is powerful enough to express all formatting that might reasonably be wished for in a manual page, the same is not true for the man(7) language. Even in a well-written man(7) manual, it is unavoidable to resort to the use of some low-level roff(7) formatting instructions.

Many lower-quality man(7) manuals, in particular those generated from other input formats by automatic translation tools, use elaborate low-level roff constructions, some of which mandoc(1) may not yet support. Most of these will show up as "unknown macros" in **mandoc −Tlint** output, and output formatting may or may not be degraded. Please report those "unknown macro" errors you find in real-world manuals to the mandoc(1) developers — unless they are mere typos in request or macro names, of course.

The remaining subsections of this section discuss particular classes of "unknown macro" errors encountered when integrating mandoc(1) into OpenBSD. The problems discussed here were solved in 2010. So if you are only interested in integrating into a new platform, and not that much in mandoc(1) architecture and development, you can safely skip these subsections.

5.3.1 Supporting the pod2man(1) preample

The pod2man(1) utility is one example of a documentation format translation program.[56] It takes documentation in the perlpod(1) format, which is an acronym for "plain old documentation", and translates it to man(7) code. Every manual page written by pod2man(1) starts with a document preamble of low-level roff(7) code. The preamble varies from one Perl version to another, but is the same for every manual written by the same Perl version.

Here is some example code from a pod2man(1) preamble:

```
$ less perl.1
[...]
.de Sp
.if t .sp .5v
.if n .sp
..
.de Vb
.ft CW
.nf
.ne \$1
..
.ds C+ C++
```

This code uses various low-level roff requests: strings definitions (.ds), macro definitions (.de), conditional requests (.if) and so on.

When integrating mandoc(1) into OpenBSD in 2010, there was no chance to get all that implemented quickly. So we went for a temporary, quick and dirty solution: We did not implement the low-level roff requests being used, but instead added explicit support for the strings and macros this low-level code defines. The strings were hardcoded, the macros implemented as if they were man(7) macros, and the rest of the low-level instructions were merely parsed an ignored. That was sufficient because the preamble is always the same.

That way, it was possible to get all man(7) manuals to work with mandoc(1), without changing any of the manuals in any way.

About half a year later, support for string and macro definitions was implemented and the temporary workarounds could be removed. So today, you won’t encounter the same issues again, but of course, you may run into other roff(7) features unimplemented in mandoc(1) that other man(7) code generators happen to use.

56. BSDCan 2011 p. 9

5.3.1.1 Case study: OpenBSD

Related timeline:
2010 Mar 01 implement pod2man(1) pseudo-macros
2010 Sep 20 mandoc can now handle the standard pod2man preamble
2010 Nov 28 remove the pod2man(1) pseudo-macros

5.3.2 The mandoc(1) tbl(7) implementation

Another area where issues might surface is in manuals using the tbl(7) language to format tables. This language is used in a relatively small number of pages, so it never received as much attention as the mdoc(7) and man(7) languages. All the same, most tables now format just fine, so you may get away without issues in this area. If you do run into unimplemented features or formatting bugs in your tree, please report to the mandoc developers.

5.3.2.1 Case study: OpenBSD

In the OpenBSD base system, only a dozen pages use tbl(7).[57] When switching the build of all other pages to mandoc, we at first continued to format this dozen with groff because mandoc did not have tbl(7) support at the time.

We long avoided the task of dealing with this handful of pages, even though Kristaps had written a stand-alone implementation of tbl(7) months before. This concerned only a very small number of manual pages, and it was unclear how to best integrate the tbl(7) parser and formatter into mandoc. Finally, I deliberately chose the minimal route: I hooked the code directly into the high-level parsers. That way, I got a working integration within a single weekend and immediately relied on it for production.

Kristaps implemented a better way later: Parse tbl(7) block macros on the roff level, then call the tbl(7) parser from the main program just like in the case of mdoc(7) and man(7). The same approach was later used for rudimentary eqn(7) support.

Related timeline:
2010 May: stand-alone implementation of tbl started by kristaps@
2010 Aug 12: OpenBSD 4.8 release rolled with mandoc
2010 Oct 15: import tbl parser and renderer written by kristaps@
2010 Oct 17: build tbl(1) pages with mandoc(1), not groff
2010 Oct 18: disconnect groff from the base build
2010 Oct 18: "I absolutely don’t intend to merge tbl into mandoc" kristaps@
2011 Jan 04: clean tbl integration by kristaps, remove mine
2011 Mar 20: rudimentary eqn support by kristaps@
2011 Jul 24: complete basic support for equation blocks

5.3.3 The design of mandoc(1) beyond mdoc(7)

The mandoc(1) program is not only intended as a formatter for the mdoc(7) language, but aims to format all contemporary man(7) manuals, too, and even historical ones back to AT&T Version 7 UNIX.

Kristaps’ original design intended to forget about the complete bottom layer and only implement the high-level mdoc(7) and man(7) macros.[58] However, since all man(7) manuals use at least some low-level roff(7) features, and many use more than they should, it turned out that does not work. There is no way around implementing some low-level roff requests.

Realizing that was a slow, painful process that dragged out over many months. Finally, we reluctantly edged in some low-level roff support. Complexity was kept to the bare minimum.

Both steps have proven to be just right: Starting from the high level gave a clean design. Our reluctance to support low-level roff(7) prevented us from getting off track. After we had put in some low-level roff support as a late addition, we were surprised how little change was required to the overall design.

57. BSDCan 2011 p. 18
58. BSDCan 2011 p. 12

The original mandoc main program looked like this:[59]

- main loop to read input files
- push line after line into the parser backends
- parsers look for high-level macros, e.g. mdoc(7)
- call the formatting frontend on the resulting syntax tree

When edging in low-level roff support, this design was changed as follows:

- main loop to read input files
- **call the roff preprocessor on each line**
- push line after line into the parser backends
- parsers look for high-level macros, e.g. mdoc(7)
- call the formatting frontend on the resulting syntax tree

So there is a stunning paradigmatic switch. Classical roff implementations first expand all high-level macros into low-level requests, then pass the low-level requests into the formatters. All structural information is lost long before the main parser. By contrast, mandoc first handles the low-level requests in a preprocessor, then passes the remaining high-level code to the parsers. Structural information is kept even if it is originally intermixed with low-level roff code.

Here is a list of roff(7) requests and their status:[60]

- mostly complete implementations in 2011:
  - string definitions: **ds**
  - macro definitions: **de rm**
  - source file inclusion: **so**
- additional mostly complete implementations today:
  - string definitions: **as tr**
  - macro definitions: **am ami dei**
  - conditionals: **if ie el ig**
  - register definitions: **nr rr**
  - input line traps: **it**
  - changing the control character: **cc**
- ignored requests:
  - adjustment: **ad ce ta**
  - spacing: **ne ns pl**
  - hyphenation: **hy nh hw**
  - character formatting: **ps fam**

The many roff requests not mentioned here are still unimplemented, but rarely occur in real-world manuals.

Related timeline:
2010 Apr 25: implement roff conditional request for man(7) only
2010 May 15: mandoc roff library started by kristaps@
2010 May 19: the roff library replaces my preliminary support for conditionals
2010 Jul 03: rudimentary implementation of user-defined strings
2010 Sep 22: interesting commit message: no hope for .de
2010 Nov 25: implement the .de request (define macro)
2010 Dec 01: implement the .so request (include source file)
2011 Jan 16: implement the .rm request (remove macro)
2011 Jul 28: implement the .tr request (translate characters)
2012 May 31: implement the \z escape sequence (zero advance)

59. BSDCan 2011 p. 13
60. BSDCan 2011 p. 14

2012 Jun 12: implement the .cc request (change control character)
2013 Mar 21: implement number registers (Christos Zoulas, NetBSD)
2013 Apr 03: support simple numerical conditions (Christos Zoulas, NetBSD)
2013 Jul 13: implement the .it request (input line trap)
2013 Dec 15: implement increment and decrement in .nr
2014 Jan 22: implement the \: escape sequence (optional line break)
2014 Feb 14: implement the .as request (append to string)
2014 Mar 08: implement string comparison in conditionals
2014 Mar 30: implement the .ll request (output line length)
2014 Apr 05: implement the .rr request (remove register)
2014 Apr 07: implement indirect references in string expansion
2014 Apr 07: almost complete implementation of numerical expressions
2014 Jul 07: implement .dei and .ami (indirect macro definitions)

5.4 Check mandoc output

For the manuals in your tree, systematically compare groff(1) and mandoc(1) output. The **gmdiff** tool in the mdocml.bsd.lv repository can help with that. To reduce noise to manageable levels, look at the patches to textproc/groff in OpenBSD ports. For example, they disable adjustment and hyphenation, make page header lines agree with mandoc, and remove the special handling of the .Pa macro in the FILES section. The comparison will not be quite easy because there are still some trivial differences, most of them regarding whitespace. That is, mandoc and groff output will not be completely identical, but try to make sure no content gets lost and no formatting is completely garbled.

If you find any serious issues, report them, in particular if mandoc(1) fails to flag them as ERRORs. Patching manuals is usually not the right approach in this phase.

Quickly move on to the next phase, that is, when you are convinced there are no show-stopper issues, not when you feel everything is perfect.[61] Your system will mature best when it's enabled by default and when you get and use real-world feedback.

5.4.1 Case study: OpenBSD

In OpenBSD, such comparisons were done as an iterative process in several cycles:[62] Comparisons were first run on a small part of the tree. Some parsing and formatting issues were identified. They were prioritized by a rough, inexact estimate of frequency and severity. Some of them were then fixed in mandoc(1): Sufficiently few that we didn't lose too much time; sufficiently many to significantly reduce the noise. Then, we started over with a larger part of the tree.

Related timeline:
2010 Aug 15 systematic bug hunting in /bin and /sbin
2011 Jan 23 systematic bug hunting in /usr/bin

5.5 Watch out for local features!

The major BSD trees forked from each other about two decades ago, and one copy of groff has dwelt in each of them for that long time. Chances are it has been locally patched at least in some way, and the local manuals rely on such patches. The manuals in a BSD tree might also use internal features of whatever groff version may be around.

When trying to switch from groff to mandoc, you will probably bump into such features because mandoc is not likely to support whatever localisms have developed in your tree.

If you can easily patch such features away in your manuals, you should probably do that because it will make your manuals more portable, not just to mandoc, but even to other breeds of groff. However, that may not be possible. In that case, please talk to the mandoc developers, and we will see what we can do to rescue your local quirks.

61. BSDCan 2011 p. 10
62. BSDCan 2011 p. 16

5.5.1 Case study: OpenBSD

Such home-grown non-standard features exist even in OpenBSD.[63] The worst one is related to SYNOPSIS formatting: An internal roff register that is specific to the pre-1.17 groff implementation is used to switch on and off SYNOPSIS mode in kernel manuals having the synopsis split into several parts. We had no reasonable choice but to support this in mandoc(1), even though what these manuals do is an incredibly dirty hack.

5.6 Switch over to build with mandoc

Once mandoc(1) is ready to build all of your tree and no show-stopper issues are left, you can do the switch to use it instead of groff for building your manuals. Choose the timing wisely. Make sure there is plenty of time until the next release. Rushing this change in shortly before a release would be a bad idea. Formatting quality of manual pages is not among the things people focus on during release testing, so chances are issues you overlooked won’t be found before release, and once the release is out of the door, you end up in a rather awkward situation that’s hard to fix.

After switching the build, watch out for bug reports from users of the -current code. Report and fix bugs as quickly as possible.

The switch itself can be done in one or two steps. OpenBSD did it in two steps: The formatter was switched from groff to mandoc in April 2010. Source manuals are installed instead of preformatted manuals since October 2010.

NetBSD did it in one step in February 2012, switching both the build system to install unformatted instead of formatted and changing the manual formatter from groff to mandoc in man.conf(5) at the same time. If you feel confident you can manage the one-step process, i recommend you follow the example of NetBSD. Nowadays, mandoc(1) is mature enough to rely on it as a run-time formatter right away, assuming you have done the necessary testing for your particular system as described in the last few sections.

5.6.1 Case study: OpenBSD

Mandoc was ready for production right before the OpenBSD 4.7 release was tagged.[64] The build was switched over right after the tree unlocked after the release, so there was as much time as possible to fix fallout. Bug reports from real-world users started coming in at once. Priority was given to these bug reports. All serious fallout fixed within a few days. Lots of time was left for polishing before the next release.

Related timeline:
2010 Mar 01: mandoc ready to build the tree
2010 Mar 18: OpenBSD 4.7 release rolled without mandoc
2010 Mar 20: Xenocara can now build with mandoc as well
2010 Apr 02: link mandoc to the OpenBSD build
2010 Apr 03: switch base build to mandoc, excepting tbl pages
2010 Apr 03: fix all fatal issues where mandoc kills ports builds
2010 Apr 04: first port maintainer explicitely switches to mandoc
2010 Apr 04: fix first mandoc bug that was found by ports usage
2010 Apr 05: espie@ implements USE_GROFF framework and groff-1.15 port
2010 Apr 07: first major merge from bsd.lv after the switch
2010 Apr 08: first mandoc bugfix found in ports propagates upstream
2010 Aug 12: OpenBSD 4.8 release rolled with mandoc

63. BSDCan 2011 p. 15
64. BSDCan 2011 p. 11

5.7 Lessons learnt from the replacement project

5.7.1 Bad patches triggering good ones

During the replacement project in OpenBSD, we put preliminary code into production on multiple occasions and later on ripped it out again.[65] That may seem inefficient at first, but actually it’s a perfectly sane approach: The first implementation explores the feature. The second implementation gets it right. Just don’t let the first one sprawl until it can’t be ripped out any more.

This approach got used in at least five cases:
2010 Mar 01 - May 14: end of sentence detection
2010 Apr 25 - May 19: roff conditionals
2010 Mar 01 - Sep 20: pod2man preamble
2010 Jun 16 - Jun 27: roff registers
2010 Oct 15 - Jan 04: tbl integration

5.7.2 Clean design works even for dirty languages

For the reasons explained in the introductory section of this paper, the mdoc(7) language is the best tool available for writing documentation, and many attempts to design something better have failed. So, mdoc(7) is an excellent tool for its job, but that doesn’t imply much about the quality of the language from a software engineering perspective.

Actually, the design of languages like mdoc(7), man(7), roff(7), tbl(7), eqn(7) isn’t the latest and greatest in software engineering. After all, the youngest of these languages is now about 25 years old, and the concepts have evolved (and required some compatibility) for another 25 years before that.

All the same, the mandoc project has shown that it is possible to design and implement a relatively clean compiler even for a set of somewhat dirty languages.

We started coding with the nice high-level stuff.[66] That gave us a very clean overall design. We edged in low-level ugliness later, only where it is required. It has proven possible to edge in low-level features even while the tool was already in production, but only because we kept all parts small and simple. In a large and complex system, changing the basic design in an afterthought would no doubt break the system.

Two reasons to shun complexity are well known: It is the enemy of correctness and security. However, here we have seen that keeping down complexity is also critical for flexibility, which is a third reason that i guess fewer people are aware of.

5.7.3 Move fast!

A replacement project has the best chances to succeed if you quickly put your work to real-world use.[67] Do not let it rot in a corner of the OS. Keep moving fast, do not fear change.

Only make sure you don’t trap your users with incompatible changes. You won’t find users when you break behaviour. And you won’t find bugs when you don’t have users.

65. BSDCan 2011 p. 23
66. BSDCan 2011 p. 24
67. BSDCan 2011 p. 25

6. Integrating mandoc as a ports documentation formatter

6.1 Handling manual pages in ports

The *Law of Feature Creep* is quite well-known in general: If a software offers some feature, sooner or later somebody will use it.[68]

It allows an easy corollary, applying it to the documentation of portable software: For every feature of the roff language (and for every groff extension), no matter how arcane and how obviously irrelevant for manual pages, sooner or later somebody will want to port a third-party software abusing that feature to format its manual pages.

That is unfortunate because mandoc(1) is not a complete nroff implementation and it is not clear that it will ever be.

In the base system of an operating system, this is not a problem: Given a finite set of manuals, all the required features can be implemented in mandoc(1), or alternatively, it is possible to patch away the worst abuse in the handful of manuals affected.

But in ports, “mandoc or nothing” is not a viable strategy: That would inevitably leave you with some seriously misformatted manuals, and in some cases with no usable manuals at all.

6.2 The OpenBSD solution for manual pages in ports

The following strategy has been designed and implemented by Marc Espie, and it has proven very sturdy and very easy to use.[69]

Make sure that no port tries to preformat manuals during the build target. Let every port install manual page sources during the fake install target.

For the majority of ports, mandoc(1) can handle all manuals: That’s it, you are done with respect to these.

For the remaining minority of ports: Set a special boolean make(1) variable in the port Makefile, in OpenBSD called USE_GROFF. That variable implies a build dependency on the groff port. When building the package, the ports framework runs groff on the fly and packages the preformatted pages instead of the source pages.[70]

After installing the packages, this will work just fine at run time: The preformatted pages will be diplayed directly by man(1), and man(1) will format the source pages with mandoc(1), with no dedicated configuration.

For example, in OpenBSD in the spring of 2014 there were 7952 ports, 1217 of which still USE_GROFF (15%). Some of these probably don’t really need it, but there is no hurry. Removing USE_GROFF needs a manual check — which was already done for about 3000 ports during the last three years. Instructions are available for checking ports in this respect.[71]

6.3 Features that help, in particular for ports

Obviously, the more manuals mandoc(1) can handle, the easier will all this become. As explained above, usage of low-level roff(7) features causes most of the problems. Consequently, the ongoing efforts to improve low-level roff(7) support help ports in particular.

The following features of this kind have been added to mandoc(1) during the last two years:[72]

- Indirect references in roff(7) expansion since April 7, 2014.

68. BSDCan 2014 p. 30
69. BSDCan 2014 p. 31
70. BSDCan 2011 p. 19
71. http://www.openbsd.org/faq/ports/specialtopics.html#Mandoc
72. BSDCan 2014 p. 37

- Expansion of roff(7) number registers since March 21, 2013 (Christos Zoulas).
- Almost complete support for roff(7) numerical expressions since April 7, 2014.
- Numeric comparison in roff(7) conditionals since April 3, 2013 (Christos Zoulas).
- String comparison in roff(7) conditionals since March 8, 2014.
- Newly supported roff(7) requests:

as	append to string
cc	change control character
it	set input line trap
ll	change output line length
rr	remove register
tr	translate characters

- newly supported **−man-ext** macros:

EX/EE	example display
OP	optional element
PD	paragraph distance
UR/UE	uniform resource identifier

6.4 makewhatis(8) in ports

Base and X manual databases are essentially static.[73] But packages get installed and deinstalled. You could wait for the periodic weekly(8) makewhatis(1) rebuild. A better idea works as follows:

- During pkg_add, run **makewhatis −d** */usr/local/man files ...*
- During pkg_delete, run **makewhatis −u** */usr/local/man files ...*

This is done routinely on OpenBSD, and it works seamlessly with the new makewhatis. So, right after pkg_add(1), you call apropos(1), and it finds the freshly installed manual pages.

6.5 Moving groff to ports

This move requires two prerequisites:

1. All manual pages in the base system need to format properly with mandoc, and man.conf(5) must have been switched over to use mandoc to format manuals by default.
2. The USE_GROFF infrastructure must be available in the ports framework, and all ports that have not been checked to work with mandoc must have the USE_GROFF Makefile variable set.[74]

6.5.1 Case study: OpenBSD

In OpenBSD, the move of groff from base to ports was done right before the 2010 ports hackathon, so there was plenty of time to deal with fallout. And indeed, it turned out that was needed.

1. With respect to item 1 above, we were late to realize that the X11 manuals use the roff **so** (file inclusion) request. It was implemented in a hurry after the move was already done, so X11 manuals were briefly broken.
2. With respect to item 2 above, the ports tree and the groff port had already been prepared by Marc Espie. Several porters immediately started moving over ports to mandoc on a case by case basis. Consequently, several bugs were reported in mandoc. Once again, we gave priority to fixing such real-world issues.

73. BSDCan 2014 p. 32
74. BSDCan 2011 p. 19

Related timeline:
2010 Aug 12: OpenBSD 4.8 release rolled with mandoc
2010 Oct 18: disconnect groff from the base build
2010 Oct 19: switch default /etc/man.conf to mandoc
2010 Oct 23: schwarze@ at p2k10 hackathon in Budapest
2010 Oct 26: support .so (low-level roff "switch source file")
2010 Oct 27: OpenBSD ports FAQ section about mandoc and groff
2010 Oct 29: landry@ performs the first major USE_GROFF removal
2010 Oct 29: millert@ removes colcrt(1), checknr(1), soelim(1)
2011 Feb 07: use mandoc in Xenocara Imake builds
2011 Mar 02: OpenBSD 4.9 release rolled without groff
2011 Mar 12: cvs rm groff
2011 Mar 19: update ports groff from 1.15 to 1.21
2011 Apr 24: tweak mandoc to conform to newest groff habits
2011 Apr 26: fixed groff-1.21 invocation for Imake ports

6.5.2 Benefits of the move

While moving groff from the base system to ports certainly brings benefits for the base system, in particular a reduction of GPL and C++ code, it also has important advantages for groff lovers:

- In the base system, you are stuck with the last groff version released under GPL v2, which is groff-1.19.2 released in 2005, but in the ports tree, GPL v3 doesn’t hurt as much.
- Even the license issue aside, it is much easier to keep software developed externally up-to-date in the ports tree than in the base system.

Newer groff has many new features, for example:

- groff-1.20 (Jan. 2009) added preconv(1) for Unicode/wide character support, several localization packages, XHTML support, a MathML output device for eqn(1), the chem(1) preprocessor for pic(1), a new hdtbl package, and many smaller features.
- groff-1.21 (Dec. 2010) added support for japanese manual pages.
- groff-1.22 (Dec. 2012) added gropdf(1) and pdfmom(1) to directly generate PDF output.

6.6 Summary: step by step instructions

1. Build and commit a textproc/groff port (version 1.22.2), even if you have groff in base.[75]
2. Introduce the USE_GROFF port Makefile variable. Trigger a groff build dependency for USE_GROFF. Implement format-on-the-fly for USE_GROFF and install formatted. Without USE_GROFF, install source manuals. Pay attention to get packing lists right.
3. Safe way: Turn USE_GROFF on for all ports having manuals. Shortcut: Skip those that don’t have it in OpenBSD.
4. Remove groff from base.
5. Start removing USE_GROFF on a case by case basis. To be extra safe: Only do it for ports having no mandoc ERRORs and identical output with groff and mandoc.
6. This is another critical phase: Stay tuned for bug reports from users and work with upstream to get them resolved.

In OpenBSD, this happened in October 2010.

75. BSDCan 2011 p. 19

7. Status and next steps

7.1 OpenBSD

7.1.1 Status

- Kristaps@ developed mandoc(1) since November 22, 2008.[76]
- Source code in the base repo since April 6, 2009.
- Schwarze@ maintaining it since June 14, 2009.
- Actively maintained regression suite since October 27, 2009.
- mandoc(1) installed with OpenBSD-current since April 2, 2010.
- Base system manuals built with mandoc(1) since April 3, 2010.
- USE_GROFF framework for ports by espie@ since April 5, 2010.
- Releases fully rely on mandoc(1) since OpenBSD 4.8, November 1, 2010.
- Groff disconnected from base build since October 18, 2010:
  mandoc(1) is the only documentation formatter in base for almost four years.
- Groff removed from the source tree since March 12, 2011.
- Groff 1.21/1.22 available from the ports tree since March 19, 2011.
- No stable releases contain groff since OpenBSD 4.9, May 1, 2011.
- Install manual sources, not preformatted manuals since June 23, 2011.
- SQLite-based code in the source tree since December 30, 2013.
- makewhatis(8)/apropos(1) using mandoc since April 18, 2014.
- New man.cgi(8) running on openbsd.org since July 12, 2014.
- All will be released with OpenBSD 5.6 on November 1, 2014.

7.1.2 Possible future directions

- Replace the traditional BSD man(1) implementation with the one from the mandoc toolbox.
- Switch the default output mode from **−Tascii** to **−Tlocale**. That does no harm when using the POSIX locale(1), and it helps people using UTF-8 locales.
- Integrate preconv(1) into mandoc(1) for better UTF-8 handling.[77]
- Improve pod2mdoc(1) to better support perlpod(1) to mdoc(7) transitions, in particular for the LibreSSL manuals.
- Support automatic semantic enrichment of Perl manuals with pod2mdoc(1). I’m not yet sure this is practicable, it’s just an idea so far.
- Support transitions from man(7) to mdoc(7) with doclifter(1) and docbook2mdoc(1).
- Unify parsers, allowing more low-level roff(7) improvements.

76. BSDCan 2014 p. 25
77. BSDCan 2014 p. 39

7.2 NetBSD

7.2.1 Status

- A pkgsrc mdocml port by Jörg Sonnenberger exists since March 1, 2009.[78]
  This implies support for *many* additional platforms.
  The current version is 1.13.1 since August 10, 2014 (Thomas Klausner).
- The first code patch was sent upstream by Jörg Sonnenberger on June 11, 2009.
- Source code in the base repo and installed by default in NetBSD-current since October 21, 2009.
- Big changes on February 7, 2012:
  - Install source manuals, no longer install preformatted manuals.
  - **Use mandoc(1) as the default run-time manual formatter instead of groff.**
  - Use makemandb(8) by Abhinav Upadhyay instead of makewhatis(8) together with versions of apropos(1) and whatis(1) based on it, featuring full text search, but not semantic search.
- All this was first released with NetBSD 6.0 on October 17, 2012.

Semantic searching is not yet supported, not even as an option.

7.2.2 Recommended next steps

- Upgrade base system version to 1.13.1.
- Integrating semantic search support is a serious problem because it is completely incompatible with the existing makemandb(8) which also uses SQLite3 but doesn't use the strengths of mdoc(7) in any way. Unfortunately, i must admit i have no idea how this dilemma might be solved.

7.3 FreeBSD

7.3.1 Status

- An mdocml port by Ulrich Spörlein exists since March 9, 2009.[79]
- First code patch sent in by Ulrich Spörlein on July 18, 2009.
- **Source code in the base repo and installed by default since October 19, 2012.**
- First released with FreeBSD 10.0 on January 20, 2014.
- mandoc(1) is installed, but not used.
- Semantic searching is not yet supported, not even as an option.

7.3.2 Recommended next steps

- Upgrade base system version to mandoc-1.13.1.
- Switch base system to use mandoc by default.

7.4 DragonFly BSD

7.4.1 Status

- **Source code in the base repo and installed by default since October 27, 2009 (Sascha Wildner)**
- First released with DragonFly BSD 3.6.0 on March 28, 2010.
- First code patch sent in by Franco Fichtner on November 25, 2013.
- mandoc(1) is installed, but not used.
- Semantic searching is not yet supported, not even as an option.

7.4.2 Recommended next steps

- Upgrade to mandoc-1.13.1.
- Switch base system to use mandoc by default.

78. BSDCan 2014 p. 26
79. BSDCan 2014 p. 27

7.5 Status in non-BSD systems

illumos
: Mandoc is contained in the base system and used by default for formatting manuals since July 21, 2014 (Garrett D’Amore). Upgraded to 1.12.3 on August 2, 2014 (Garrett D’Amore).

Minix 3
: The mandoc source code is in the base repository since June 26, 2010 (Ben Gras).[80] However, the project appears to be somewhat apathetic. It is still using a version that is more than four years old.

Alpine Linux
: An aport exists since July 6, 2010 (Natanael Copa). It moved from testing to main on June 12, 2011 (Natanael Copa). It is continuously maintained.

Arch Linux
: An mdocml package exists since October 3, 2010 (Markus M. May). It was updated to 1.12.3 on April 17, 2014 (new maintainer Jesse Adams) and to 1.13.1 on August 17, 2014 (Jesse Adams).

Slackware Linux
: An mdocml package exists since January 7, 2014 (Dániel Lévai).

CentOS, Debian, Fedora, RedHat, SuSE, Ubuntu Linux
: Unofficial mdocml packages exists since April 19, 2014 (Jesse Adams).

MacOS X
: An mdocml package exists since September 5, 2010, but it seems abandoned.

Cygwin
: An mdocml package exists since December 12, 2012 (Yaakov Selkowitz). It was updated to 1.12.2 on November 11, 2013 (Yaakov Selkowitz).

80. BSDCan 2014 p. 28

8. Exercises

The following pages propose some exercises for the hands-on working phases of the tutorial.

Before starting to work, consider the following points:

1. The amount of exercises proposed is intentionally *much* bigger than what anybody can possibly handle in the 40 minutes available for working on them in this tutorial, such that you have a *choice* both with respect of topic area and difficulty.
2. Do not waste time studying all the instructions of all the exercises. Look at the table of contents for topic areas you are interested in, then skim the instruction texts of exercises for that area. Only study those instructions in detail you actually consider working on.
3. Feel free to work alone or with one or two partners at your choice.
4. Feel free to ask any questions you might have to me or any other participant during the working phases. Do not fear interrupting people. Everybody can work in a concentrated manner for hours on end at home. What you can have here and what's harder to get at home is *cooperation*.
5. In the first, longer (30 minutes) phase, feel free to spend all the time on one complex task or explore several - probably not more than three - smaller tasks. In the second, shorter (10 minutes) phase, you should probably focus on one task, or at most two. Try to avoid time-consuming repetitive work during the tutorial. For example, instead of writing down a long of list options, just write down the first two or three, then move on to the next section.
6. If, during your work, you discover anything you consider worth presenting to the other participants of the tutorial, talk to me during the working phase, and i may be able to reserve a slot of one to three minutes for you. Nobody will be forced to present their results, though.

8.0 Recommended exercises

The following exercises are recommended for the following target audiences. Of course, all participants are free to choose any exercise they are interested in. See the table of contents at the very end of this document.

The exercises marked "second phase" are easier to handle in the second working phase because the subject will only be explained *after* the first working phase. If you have a strong interest in some exercise marked "second phase" and feel able to handle it, there is nothing wrong with attempting it in the (longer) first working phase, though.

Software or documentation developers...

... working on software lacking one or more manual pages:
first phase: Exercise 8.1.1, page 41.

... working on software having non-mdoc documentation:
first phase: Exercise 8.1.2, page 41.

... working on software having mdoc documentation:
first phase: Exercise 8.3.1, page 42.

... maintaining a portable software package:
first phase: Exercise 8.2.1, page 42. Exercise 8.2.2, page 42.

Porters and port maintainers:
first phase: Exercise 8.3.1, page 42. Exercise 8.3.2, page 43.
second phase: Exercise 8.4.1, page 43. Exercise 8.4.2, page 43.

Operating system developers and documentation maintainers:
first phase: Exercise 8.3.2, page 43.
second phase: Exercise 8.4.1, page 43. Exercise 8.4.2, page 43.

System administrators:
  second phase: Exercise 8.4.2, page 43.

End users:
  second phase: Exercise 8.4.1, page 43.

8.1 Using the mdoc(7) formatting language

8.1.1 Writing a manual from scratch

If there is a software, ideally not too complex and with a limited number of features, that you maintain or just care about and that doesn't have any manual page yet, start writing a manual page for it using the mdoc(7) language.

Hints:

- Be careful to not get lost in details.
- Start by setting up the preamble and *all* required section titles.
- In each section, take rough notes which material will need to be covered in that section. Only after that, start filling in individual sections.
- Pay particular attention to get the beginning of the DESCRIPTION concise and clear. It should clearly state the purpose of the software and not be wordy.
- When drafting lists, first figure out which list entries will be needed, only then start filling them in.
- If there is more material to cover than you can finish during the tutorial, don't fear leaving blanks in the middle. Try to work on as many different parts as possible and take advantage of the chance to ask questions that arise.
- Consider submitting the result to the software maintainer for inclusion, possibly after finishing it at home if time is insufficient during the tutorial.

8.1.2 Translating a manual to mdoc

If there is a software, ideally not too complex and with a limited number of features, that you maintain or just care about and that has a manual in a language different from mdoc(7), start translating the manual to mdoc(7). Working on this exercise is particularly useful if you have reason to believe that the maintainer might be willing to switch the format.

Hints:

- Be sure to keep a copy of the original.
- If there is more material to cover than you can finish during the tutorial, don't fear leaving blanks in the middle. Try to work on as many different parts as possible and take advantage of the chance to ask questions that arise.
- If the original was a man(7) page, compare the output of mandoc(1) from your version with the output **mandoc −Omdoc** produces from the original version.
- Consider submitting the result to the software maintainer for inclusion, possibly after finishing it at home if time is insufficient during the tutorial.

8.2 Manual pages for portable software

8.2.1 Package autogenerated man and cat

Take a portable software package you maintain or care about that already has mdoc(7) manuals or where you consider converting the manuals to mdoc(7). Design and implement make(1) targets in the distribution tarball build system to generate and package man(7) and cat (preformatted) versions of the existing mdoc(7) manuals.

Caveat:

The difficulty of this exercise, and the amount of work required, can vary greatly depending on the size and complexity of the software package and the build system used. For a large or complex package, familiarity with the build system is probably required.

Hints:

- You must have a checked out copy of the source repository. This exercise cannot be solved starting from a mere release tarball.
- If the mdoc(7) versions of the pages have not yet been written, don’t waste your time on that, just write two or three ten-line stubs to get started.
- Pay attention to not edit autogenerated files, like *Makefile* if there is also *Makefile.in*.
- Try to use make(1) inference (suffix) transformation rules. Avoid writing one rule for each manual page if possible.
- To test your changes, build and inspect a release tarball.

8.2.2 Write configuration tests

Take a portable software package you maintain or care about that already has mdoc(7) manuals or where you consider converting the manuals to mdoc(7). Design and implement configuration tests to decide whether to install mdoc, man, or cat manuals on the target system.

Caveat:

The difficulty of this exercise, and the amount of work required, can vary greatly depending on which build system is used and how it is used. You better have a at least a rough idea what you are doing and some experience with the build system in question, or at least with build systems in general, before attempting this exercise.

Hints:

- If possible, check out a copy of the source repository of the software to work in. Depending on the build system used, merely having a release tarball may or may not be sufficient to do meaningful work on this exercise.
- Pay attention to not edit autogenerated files, like *./configure* in software using GNU autoconf(1).
- An easy test of your changes can probably be done in the checkout area by running the tool to regenerate autogenerated files, then running the configuration script and inspect what it detects.
- The ultimate test would be to build a release tarball, than test from that.

8.3 Quality control for existing manuals

8.3.1 Checking one or a few specific pages

Choose a small number of manual pages you maintain or care about. Run those check tools — mandoc(1) **−Tlint**, mdoclint(1), igor(1), and maybe gmdiff — you have installed or can easily install on them. Prioritize your findings and take notes about those you consider relevant. If desired, prepare patches to improve the manuals.

Ask somebody who is experienced with mdoc(7) to have a look at your patches, then send them upstream for inclusion.

8.3.2 Running bulk quality checks

In a larger tree of manuals you maintain or care about, run **mandoc −Tlint −Wfatal**, solve any issues showing up, run **mandoc −Tlint −Werror**, and start working on the issues found.

Hints:

- Don't be shy to use find(1) or scripting as required to quickly and efficiently get at the files you need. On the other hand, don't waste your time devising elaborate frameworks right now; this tutorial is about manuals, not about build systems!
- Refrain from large-scale patching until you are really sure what you see are large-scale errors in your tree and cannot reasonably be solved in other ways, for example by improving mandoc(1). For this exercise, communication is of particular importance, there *are* people around you can talk to!
- If you arrive in a state where the are no more errors, you can proceed to look at warnings.

8.4 Searching and displaying manual pages

8.4.1 Testing makewhatis and apropos

Choose a manual tree you are maintaining or care about. Run makewhatis(8) on it. Try out various apropos(1) features and try to discover issues. Report any you find to me.

Hints:

- For this exercise, having an up-to-date mandoc suite is more important than for most. Consider checking out from anoncvs@mdocml.bsd.lv:/cvs and building and installing from that before starting. On OpenBSD, it is better to build from */usr/src/usr.bin/mandoc* instead.
- To run makewhatis(8) on one specific tree only, you will need the *dir* argument.
- If everythings appears to be working fine, try to refine your methods. For example, try the **−p**, **−D**, and **−DD** options of makewhatis(8) and try to understand the output.
- Probably, most of the output will be false positives, but you may also find some real bugs in the manuals in this way that can be worth patching.
- If you are comfortable with SQL, run the sqlite3(1) command line client directly on the database generated by makewhatis(8) and look for anything that seems questionable.

8.4.2 Checking manual locations and formats

On your favourite operating system, figure out all locations where manuals are installed and identify those that man(1) searches by default. Figure out all formats of installed manual pages (formatted/unformatted, source language, compression, whatever). Check that makewhatis(8) and apropos(1) work for all locations and all formats. Report any issues you find to me.

Hints:

- For this exercise, having an up-to-date mandoc suite is more important than for most. Consider checking out from anoncvs@mdocml.bsd.lv:/cvs and building and installing from that before starting. On OpenBSD, it is better to build from */usr/src/usr.bin/mandoc* instead.

CONTENTS