5 gtroff Reference

This chapter covers *all* of the facilities of the GNU `troff` formatting engine. Users of macro packages may skip it if not interested in details.

5.1 Text

AT&T `troff` was designed to take input as it would be composed on a typewriter, including the teletypewriters used as early computer terminals, and relieve the user of having to be concerned during the drafting process with the precise line length that the final version of the document would use, where words should be hyphenated, and how to achieve straight margins on both the left and right sides of the page. Early in its development, the program gained the ability to prepare output for a phototypesetter; a document could then be prepared for output to either a teletypewriter, a phototypesetter, or both. GNU `troff` continues this tradition of permitting an author to compose a single master version of a document which can then be rendered for a variety of output formats or devices.

`roff` input files contain text interspersed with instructions to control the formatter. Even in the absence of such instructions, GNU `troff` still processes its input in several ways, by filling, hyphenating, breaking, and adjusting it, and supplementing it with inter-sentence space.

5.1.1 Filling

When GNU `troff` starts up, it obtains information about the device for which it is preparing output.[1] A crucial example is the length of the output line, such as “6.5 inches”.

GNU `troff` reads its input character by character, collecting words as it goes, and fits as many words together on an output line as it can—this is known as *filling*. To GNU `troff`, a *word* is any sequence of one or more characters that aren’t spaces, tabs, or newlines. Words are separated by spaces, tabs, newlines, or file boundaries.[2] To disable filling, see Section 5.7 [Manipulating Filling and Adjustment], page 87.

```
It is a truth universally acknowledged
that a single man in possession of a
good fortune must be in want of a wife.
    ⇒ It is a truth universally acknowledged that a
    ⇒ single man in possession of a good fortune must
    ⇒ be in want of a wife.
```

[1] See Section 8.2 [Device and Font Files], page 232.

[2] There are also *escape sequences* which can function as word characters, word separators, or neither—the last simply have no effect on GNU `troff`’s idea of whether its input is within a word or not.

5.1.2 Sentences

A passionate debate has raged for decades among writers of the English language over whether more space should appear between adjacent sentences than between words within a sentence, and if so, how much, and what other circumstances should influence this spacing.[3] GNU `troff` follows the example of AT&T `troff`; it attempts to detect the boundaries between sentences, and supplies additional inter-sentence space between them.

```
Hello, world!
Welcome to groff.
    ⇒ Hello, world!  Welcome to groff.
```

GNU `troff` does this by flagging certain characters (normally '`!`', '`?`', and '`.`') as *end-of-sentence* characters; when GNU `troff` encounters one of these characters at the end of a line, or one of them is followed by two spaces on the same input line, it appends an inter-word space followed by an inter-sentence space in the formatted output.

```
R. Harper subscribes to a maxim of P. T. Barnum.
    ⇒ R. Harper subscribes to a maxim of P. T. Barnum.
```

In the above example, inter-sentence space is not added after '`P.`' or '`T.`' because the periods do not occur at the end of an input line, nor are they followed by two or more spaces. Let's imagine that we've heard something about defamation from Mr. Harper's attorney, recast the sentence, and reflowed it in our text editor.

```
I submit that R. Harper subscribes to a maxim of P. T.
Barnum.
    ⇒ I submit that R. Harper subscribes to a maxim of
    ⇒ P. T.  Barnum.
```

"Barnum" doesn't begin a sentence! What to do? Let us meet our first *escape sequence*, a series of input characters that give instructions to GNU `troff` instead of being copied as-is to output device glyphs.[4] An escape sequence begins with the backslash character `\` by default, an uncommon character in natural language text, and is *always* followed by at least one other character, hence the term "sequence".

The non-printing input break escape sequence `\&` can be used after an end-of-sentence character to defeat end-of-sentence detection on a per-instance basis. We can therefore rewrite our input more defensively.

[3] A well-researched jeremiad appreciated by `groff` contributors on both sides of the sentence-spacing debate can be found at `https://web.archive.org/web/20171217060354/http://www.heracliteanriver.com/?p=324`.

[4] This statement oversimplifes; there are escape sequences whose purpose is precisely to produce glyphs on the output device, and input characters that *aren't* part of escape sequences can undergo a great deal of processing before getting to the output.

```
I submit that R.\& Harper subscribes to a maxim of P.\&
T.\& Barnum.
    ⇒ I submit that R. Harper subscribes to a maxim of
    ⇒ P. T. Barnum.
```

Adding text caused our input to wrap; now, we don't need the escape after '`T.`' but we do after '`P.`'. Ensuring that potential sentence boundaries are robust to editing activities and reliably understood both by GNU `troff` and the document author is a goal of the advice presented in Section 5.1.10 [Input Conventions], page 67.

Normally, the occurrence of a visible non-end-of-sentence character (as opposed to a space or tab) immediately after an end-of-sentence character cancels detection of the end of a sentence. For example, it would be incorrect for GNU `troff` to infer the end of a sentence after the dot in '`3.14159`'. However, several characters are treated *transparently* after the occurence of an end-of-sentence character. That is, GNU `troff` does not cancel end-of-sentence detection when it processes them. This is because such characters are often used as footnote markers or to close quotations and parentheticals. The default set is '`"`', '`'`', '`)`', '`]`', '`*`', `\[dg]`, `\[dd]`, `\[rq]`, and `\[cq]`. The last four are examples of *special characters*, escape sequences whose purpose is to obtain glyphs that are not easily typed at the keyboard, or which have special meaning to GNU `troff` (like \ itself).[5]

```
\[lq]The idea that the poor should have leisure has always
been shocking to the rich.\[rq]
(Bertrand Russell, 1935)
    ⇒ "The idea that the poor should have
    ⇒ leisure has always been shocking to
    ⇒ the rich."  (Bertrand Russell, 1935)
```

The sets of characters that potentially end sentences or are transparent to sentence endings are configurable. See the `cflags` request in Section 5.17.4 [Using Symbols], page 125. To change the additional inter-sentence spacing amount—even to remove it entirely—see Section 5.7 [Manipulating Filling and Adjustment], page 87.

5.1.3 Hyphenation

When an output line is nearly full, it is uncommon for the next word collected from the input to exactly fill it—typically, there is room left over only for part of the next word. The process of splitting a word so that it appears partially on one line (with a hyphen to indicate to the reader that the word has been broken) with the remainder of the word on the next is *hyphenation*. Hyphenation points can be manually specified; GNU `troff` also uses a hyphenation algorithm and language-specific pattern files (based on those used in TEX) to decide which words can be hyphenated and where.

[5] The mnemonics for the special characters shown here are "dagger", "double dagger", "right (double) quote", and "closing (single) quote". See the *groff_char*(7) man page.

Hyphenation does not always occur even when the hyphenation rules for a word allow it; it can be disabled, and when not disabled there are several parameters that can prevent it in certain circumstances. See Section 5.8 [Manipulating Hyphenation], page 92.

5.1.4 Breaking

Once an output line has been filled, whether or not hyphenation has occurred on that line, the next word read from the input will be placed on a different output line; this is called a *break*. In this manual and in `roff` discussions generally, a "break" if not further qualified always refers to the termination of an output line. When the formatter is filling text, it introduces breaks automatically to keep output lines from exceeding the configured line length. After an automatic break, GNU `troff` adjusts the line if applicable (see below), and then resumes collecting and filling text on the next output line.

Sometimes, a line cannot be broken automatically. This usually does not happen with natural language text unless the output line length has been manipulated to be extremely short, but it can with specialized text like program source code. We can use `perl` at the shell prompt to contrive an example of failure to break the output line. The regular output is omitted below.

```
$ perl -e 'print "#" x 80, "\n";' | nroff
    error  warning: can't break line
```

The remedy for these cases is to tell GNU `troff` where the line may be broken without hyphens. This is done with the non-printing break point escape sequence '`\:`'; see Section 5.8 [Manipulating Hyphenation], page 92.

What if the document author wants to stop filling lines temporarily, for instance to start a new paragraph? There are several solutions. A blank input line not only causes a break, but by default it also outputs a one-line vertical space (effectively a blank output line). This behavior can be modified; see Section 5.24.3 [Blank Line Traps], page 175. Macro packages may discourage or disable the blank line method of paragraphing in favor of their own macros.

A line that begins with one or more spaces causes a break. The spaces are output at the beginning of the next line without being *adjusted* (see below); however, this behavior can be modified (see Section 5.24.4 [Leading Space Traps], page 176). Again, macro packages may provide other methods of producing indented paragraphs. Trailing spaces on text lines are ignored.[6]

What if there is no next input word? Or the file ends before enough words have been collected to fill an output line? The end of the file causes a break, resolving both of these cases. Certain requests also cause breaks, implicitly or explicitly. This is discussed in Section 5.7 [Manipulating Filling and Adjustment], page 87.

---

[6] "Text lines" are defined in Section 5.1.7 [Requests and Macros], page 63.

5.1.5 Adjustment

After GNU `troff` performs an automatic line break, it then tries to *adjust* the line: inter-word spaces are widened until the text reaches the right margin. Extra spaces between words are preserved. Leading and trailing spaces are handled as noted above. Text can be aligned to the left or right margins or centered; see Section 5.7 [Manipulating Filling and Adjustment], page 87.

5.1.6 Tab Stops

GNU `troff` translates horizontal tab characters, also called simply "tabs", in the input into movements to the next tab stop. These tab stops are by default located every half inch measured from the current position on the input line. With them, simple tables can be made.[7] However, this method can be deceptive, as the appearance (and width) of the text in an editor and the results from GNU `troff` can vary greatly, particularly when proportional typefaces are used.

A tab character does not cause a break and therefore does not interrupt filling. We use an arrow → below to indicate an input tab character.

```
1
→ 2 → 3 → 4
→ → 5
⇒ 1          2          3          4                       5
```

GNU `troff` provides sufficient facilities for sophisticated table composition; see Section 5.10 [Tabs and Fields], page 102. There are many details to track when using such low-level features, so most users turn to the *tbl*(1) preprocessor for table construction.

5.1.7 Requests and Macros

We have now encountered almost all of the syntax there is in `roff` languages, with an exception already noted several times in passing. A *request* is an instruction to the formatter that occurs after a control character. A *control character* must occur at the beginning of an input line to be recognized.[8] The regular control character has a counterpart, the *no-break control character*, which suppresses the break that is implied by some requests. The default control characters are the dot (`.`) and the neutral apostrophe (`'`), the latter being the no-break control character. These characters were chosen because it is uncommon for lines of text in natural languages to begin with periods or apostrophes. If you require a literal period or neutral apostrophe where GNU `troff` is expecting a control character, prefix it with the non-printing input break escape sequence, '`\&`'.

[7] "Tab" is short for "tabulation", revealing the term's origin as a spacing mechanism for table arrangement.

[8] A control character is also expected in arguments to the `if`, `ie`, `el`, and `while` requests.

An input line beginning with a control character is called a *control line*. Every line of input that is not a control line is a *text line*.[9]

Requests often take *arguments*, words (separated from the request name and each other by spaces) that specify details of the action GNU `troff` is expected to perform. If a request is meaningless without arguments, it is typically ignored.

GNU `troff`'s requests and escape sequences comprise the control language of the formatter. Of key importance are the requests that define macros. Macros are invoked like requests, enabling the request repertoire to be extended or overridden.[10]

A *macro* can be thought of as an abbreviation you can define for a collection of control and text lines. When the macro is *called* by giving its name after a control character, it is replaced with what it stands for. The process of textual replacement is known as *interpolation*.[11] Interpolations are handled as soon as they are recognized, and once performed, a `roff` formatter scans the replacement for further requests, macro calls, and escape sequences.

In `roff` systems, the `de` request defines a macro.[12]

```
.de DATE
2020-11-14
..
```

The foregoing input produces no output by itself; all we have done is store some information. Observe the pair of dots that ends the macro definition. This is a default; you can specify your own terminator for the macro definition as the second argument to the `de` request.

```
.de NAME ENDNAME
Heywood Jabuzzoff
.ENDNAME
```

In fact, the ending marker is itself the name of a macro that will be called if it is defined at the time the macro definition begins.

```
.de END
Big Rip
..
.de START END
Big Bang
.END
.START
    ⇒ Big Rip Big Bang
```

[9] The `\RET` escape sequence can alter how an input line is classified; see Section 5.14 [Line Control], page 115.

[10] Argument handling in macros is more flexible but also more complex. See Section 5.5.1.1 [Request and Macro Arguments], page 76.

[11] Some escape sequences undergo interpolation as well.

[12] GNU `troff` offers additional ones. See Section 5.21 [Writing Macros], page 154.

In the foregoing example, "Big Rip" printed before "Big Bang" because its macro was *called* first. Consider what would happen if we dropped `END` from the '`.de START`' line and added `..` after `.END`. Would the order change?

Let us consider a more elaborate example.

```
.de DATE
2020-10-05
..
.
.de BOSS
D.\& Kruger,
J.\& Peterman
..
.
.de NOTICE
Approved:
.DATE
by
.BOSS
..
.
Insert tedious regulatory compliance paragraph here.

.NOTICE

Insert tedious liability disclaimer paragraph here.

.NOTICE
    ⇒ Insert tedious regulatory compliance paragraph here.
    ⇒
    ⇒ Approved: 2020-10-05 by D. Kruger, J. Peterman
    ⇒
    ⇒ Insert tedious liability disclaimer paragraph here.
    ⇒
    ⇒ Approved: 2020-10-05 by D. Kruger, J. Peterman
```

The above document started with a series of lines beginning with the control character. Three macros were defined, with a `de` request declaring each macro's name, and the "body" of the macro starting on the next line and continuing until a line with two dots '`..`' marked its end. The text proper began only after the macros were defined; this is a common pattern. Only the `NOTICE` macro was called "directly" by the document; `DATE` and `BOSS` were called only by `NOTICE` itself. Escape sequences were used in `BOSS`, two levels of macro interpolation deep.

The advantage in typing and maintenance economy may not be obvious from such a short example, but imagine a much longer document with dozens of such paragraphs, each requiring a notice of managerial approval. Consider what must happen if you are in charge of generating a new version of such a document with a different date, for a different boss. With well-chosen macros, you only have to change each datum in one place.

In practice, we would probably use strings (see Section 5.19 [Strings], page 142) instead of macros for such simple interpolations; what is important here is to glimpse the potential of macros and the power of recursive interpolation.

We could have defined `DATE` and `BOSS` in the opposite order; perhaps less obviously, we could also have defined them *after* `NOTICE`. "Forward references" like this are acceptable because the body of a macro definition is not (completely) interpreted, but stored instead (see Section 5.21.1 [Copy Mode], page 157). While a macro is being defined (or appended to), requests are not interpreted and macros not interpolated, whereas some commonly used escape sequences *are* interpolated. `roff` systems also support recursive macros—as long as you have a way to break the recursion (see Section 5.20 [Conditionals and Loops], page 148). For maintainable `roff` documents, arrange your macro definitions so that they are most easily understood when read from beginning to end.

5.1.8 Macro Packages

Macro definitions can be collected into *macro files*, `roff` input files designed to produce no output themselves but instead ease the preparation of other `roff` documents. There is no syntactical difference between a macro file and any other `roff` document; only its purpose distinguishes it. When a macro file is installed at a standard location and suitable for use by a general audience, it is often termed a *macro package*.[13] Macro packages can be loaded by supplying the `-m` option to `groff` or `troff`. Alternatively, a `groff` document wishing to use a macro package can load it with the `mso` ("macro source") request.

5.1.9 Input Encodings

The `groff` front end calls the `preconv` preprocessor to handle most input character encoding issues without troubling the user. Direct input to GNU `troff`, on the other hand, must be in one of two encodings it can recognize.

`cp1047`
: The code page 1047 input encoding works only on EBCDIC platforms (and conversely, the other input encodings don't work with EBCDIC); the file `cp1047.tmac` is loaded at start-up.

`latin1`
: ISO Latin-1, an encoding for Western European languages, is the default input encoding on non-EBCDIC platforms; the file `latin1.tmac` is loaded at start-up.

Any document that is encoded in ISO 646:1991 (a descendant of USAS X3.4-1968 or "US-ASCII"), or, equivalently, uses only code points from the

[13] Macro files and packages frequently define registers and strings as well.

"C0 Controls" and "Basic Latin" parts of the Unicode character set is also a valid ISO Latin-1 document; the standards are interchangeable in their first 128 code points.[14]

The remaining encodings require support that is not built-in to the GNU `troff` executable; instead, they use macro packages.

`latin2` To use ISO Latin-2, an encoding for Central and Eastern European languages, either use '`.mso latin2.tmac`' at the very beginning of your document or use '`-mlatin2`' as a command-line argument to `groff`.

`latin5` To use ISO Latin-5, an encoding for the Turkish language, either use '`.mso latin5.tmac`' at the very beginning of your document or use '`-mlatin5`' as a command-line argument to `groff`.

`latin9` ISO Latin-9 is a successor to Latin-1. Its main difference from Latin-1 is that Latin-9 contains the Euro sign. To use this encoding, either use '`.mso latin9.tmac`' at the very beginning of your document or use '`-mlatin9`' as a command-line argument to `groff`.

Some characters from an input encoding may not be available with a particular output driver, or their glyphs may not have representation in the font used. For terminal devices, fallbacks are defined, like '`EUR`' for the Euro sign and '`(C)`' for the copyright sign. For typesetter devices, it usually suffices to install fonts that contain the necessary glyphs and have compatible metrics with other fonts used in the document.

Due to the importance of the Euro glyph in Europe, `groff` is distributed with a PostScript font called `freeeuro.pfa`, which provides various glyph shapes for the Euro. Thus, the Latin-9 encoding is supported for the `ps` and `pdf` drivers out of the box, while Latin-2 is is not.

Unicode supports characters from all other input encodings; the `utf8` output driver for terminals therefore does as well. The DVI output driver supports both the Latin-2 and Latin-9 encodings if the command-line option `-mec` is used as well.[15]

5.1.10 Input Conventions

Since GNU `troff` fills text automatically, it is common practice in `roff` languages to not attempt careful visual composition of text in input files: it is the esthetic appeal of the formatted output that matters. Therefore, `roff` input should be arranged such that it is easy for authors and maintainers to compose and develop the document, understand the syntax of `roff` requests,

[14] The *semantics* of certain punctuation code points have gotten stricter with the successive standards, a cause of some frustration among man page writers; see the *groff_char*(7) man page.

[15] The DVI output device defaults to using the Computer Modern (CM) fonts; `ec.tmac` loads the EC fonts instead, which have greater code point coverage.

macro calls, and preprocessor languages used, and predict the behavior of the formatter. Several traditions have accrued in service of these goals.

- Break input lines after sentence-ending punctuation to ease their recognition (see Section 5.1.2 [Sentences], page 60). It is frequently convenient to break after colons and semicolons as well, as these typically precede independent clauses. Consider breaking after commas; they often occur in lists that become easy to scan when itemized by line, or constitute supplements to the sentence that are added, deleted, or updated to clarify it. Parenthetical and quoted phrases are also good candidates for placement on input lines by themselves.
- Set your text editor's line length to 72 characters or fewer.[16] This limit, combined with the previous advice regarding breaking around punctuation, makes it less common that an input line will wrap in your text editor, and thus will help you perceive excessively long constructions in your text. Recall that natural languages originate in speech, not writing, and that punctuation is correlated with pauses for breathing and changes in prosody.
- Use `\&` after '`!`', '`?`', and '`.`' if they are followed by space, tab, or newline characters and don't end a sentence.
- Do not attempt to format the input in a WYSIWYG manner (i.e., don't try using spaces to get proper indentation or align columns of a table).
- Comment your document. It is never too soon to apply comments to record information of use to future document maintainers (including your future self). We thus introduce another escape sequence, `\"`, which causes GNU `troff` to ignore the remainder of the input line.
- Use the empty request—a control character followed immediately by a newline—to visually manage separation of material in input files. The `groff` project's own documents use an empty request between sentences, after macro definitions, and where a break is expected, and two empty requests between paragraphs or other requests or macro calls that will introduce vertical space into the document.

  You can combine the empty request with the comment escape to include whole-line comments in your document, and even "comment out" sections of it.

We conclude this section with an example sufficiently long to illustrate most of the above suggestions in practice. For the purpose of fitting the example between the margins of this manual with the font used for its typeset version, we have shortened the input line length to 58 columns. As before, an arrow → indicates a tab character.

[16] Emacs: `fill-column: 72`; Vim: `textwidth=72`

```
.\" raw roff input example
.\"   nroff this_file.roff | less
.\"   groff this_file.roff > this_file.ps
→The theory of relativity is intimately connected with the
theory of space and time.
.
I shall therefore begin with a brief investigation of the
origin of our ideas of space and time,
although in doing so I know that I introduce a
controversial subject.
.
.\" remainder of paragraph elided
.
.

→The experiences of an individual appear to us arranged in
a series of events;
in this series the single events which we remember appear
to be ordered according to the criterion of
\[lq]earlier\[rq] and \[lq]later\[rq], \" punct swapped
which cannot be analysed further.
.
There exists,
therefore,
for the individual,
an I-time,
or subjective time.
.
This itself is not measurable.
.
I can,
indeed,
associate numbers with the events,
in such a way that the greater number is associated with
the later event than with an earlier one;
but the nature of this association may be quite arbitrary.
.
This association I can define by means of a clock by
comparing the order of events furnished by the clock with
the order of a given series of events.
.
We understand by a clock something which provides a series
of events which can be counted,
and which has other properties of which we shall speak
later.
.\" Albert Einstein, _The Meaning of Relativity_, 1922
```