

tess User Manual

Less-style terminal pager with structured-log filtering
and pretty-printing

0.58.0

2026

Thomas Björk

Contents

tess — User Manual	1
Synopsis	1
Quick start	1
Command-line flags	1
Display	1
Source	2
Character encoding	3
Structured logs	4
Runtime filtering & per-pane predicates	5
Pretty-printing	6
Batch (non-interactive) output	6
Clipboard	7
Other	7
Interactive keys	8
Mouse capture	9
Horizontal scrolling	10
Split view	11
Per-pane flags: the -- form	11
Horizontal (stacked) split	12
Synchronized scrolling (:scrolllock)	13
Diff mode (--diff / :diff)	13
Git diff (--gitdiff)	14
Search	14
Incremental search (--incsearch)	15
Case sensitivity	15
Suppressing match highlights	15
Skipping the screen on search (-a)	15
Tildes past end-of-file (--tilde)	16
Startup commands (+CMD)	16
Exit-on-EOF and one-screen pager	16
Display tweaks	16
Skipping the alt-screen (-x)	17
Option-toggle prefix (-)	17
Status line	17
Customizing the status line	18
Customizing key bindings	19
Log formats	20
Built-in formats	20
Defining your own	20
Nested capture groups	21
Display templates	21
How filtering works	22
Global and local config	22
Merge semantics	23
Error handling	23
Visibility	23

Brew installs (macOS)	23
Marks	23
Multi-file navigation	24
State preserved across file switches	25
State reset on every switch	25
Stdin and multi-file	25
--follow and --live with multi-file	25
Tag jumping	25
Loading a tags file	25
Jumping	25
Multi-match cycling	25
Supported tag-file formats	26
Custom prompt placement	26
Completion	26
Auto-reload	26
:tselect — picker for multi-match tags	26
Chained ; addresses	26
Color and ANSI escapes	27
Opting out	27
Switching modes at runtime	27
Truecolor (24-bit RGB)	27
Theming the status line and prompt	27
Embedding SGR in display / prompt templates	28
What patterns see	28
Color state across lines	28
Output to non-TTY	28
Running shell commands	28
Hex display	29
Images	29
Flags	29
Animation	30
Color and export	30
Cargo feature	31
Preprocessing input	31
Multi-line records	31
Examples	32
Plain file viewing	32
Piped input	32
Quick first/last N	32
Following live output	33
Apache log analysis	33
Custom format from your own app	33
Filtering piped output	34
Combining everything	34
tess --list-formats	34
Groups: command-line shortcuts	35
Example	35

Group fields	35
Named OR-groups in config	36
Override semantics	37
Restrictions	37
Layouts: saved split arrangements	37
Pane fields	38
Rules and restrictions	38
Files	38
Exit codes	38
Common pitfalls	38
Glossary	39
Versions	39

tess — User Manual

A `less`-style terminal pager with structured-log support. macOS + Linux.

Synopsis

```
tess [OPTIONS] [FILE]...  
cmd | tess [OPTIONS]
```

`tess` opens a file or piped stdin and lets you scroll, page, search, follow, and (with a log format declared) filter by parsed fields.

Quick start

Goal	Command
View a file	<code>tess Cargo.toml</code>
View piped output	<code>git log tess</code>
Watch a log live	<code>tess -f /var/log/syslog</code>
Watch a file get rewritten (e.g. by an editor)	<code>tess --live src/main.rs</code>
Pretty-print a JSON / YAML / etc. file	<code>tess --prettify config.json</code>
Force a content type when the extension is unhelpful	<code>tess --content-type=json data.bin</code>
Show line numbers	<code>tess -N script.sh</code>
Don't wrap long lines	<code>tess -S /etc/hosts</code>
Show last 1000 lines	<code>tess --tail 1000 huge.log</code>
Tail-follow last 1000	<code>tess -f --tail 1000 huge.log</code>
Show only first 50	<code>tess --head 50 file.txt</code>
Apache 5xx errors	<code>tess --format apache-combined --filter status~^5 access.log</code>
Filter to a file (non-interactive)	<code>tess --format apache-combined --filter status~^5 -o errors.log access.log</code>
Pretty-print a file to stdout	<code>tess --prettify --stdout config.json</code>
Reformat each line through a template	<code>tess --format apache-combined --display '[<status>] <method> <url>' access.log</code>

Command-line flags

Display

- `-N`, `--LINE-NUMBERS` — show line numbers in a left-side gutter.
- `-J`, `--status-column` — add a one-column gutter at the far left edge. On a marked line it shows the mark's letter; otherwise, on a line containing a current-search match, it shows `*` (a mark beats `*`). The gutter is fixed under horizontal scroll and is drawn only on the first wrap-row of a wrapped line. No-op in `--hex`, `-r`, and image modes. Match detection runs on the line *content* only (never the gutter) and works even when `-G` suppresses the visual reverse-video highlight. Mirrors `less -J`.

- **-S**, **--chop-long-lines** — truncate long lines at the right edge instead of wrapping. Toggle interactively with **Shift-S**.
- **--tab-width N** — tab-stop width (default 8).
- **-x LIST**, **--tabs LIST** — set explicit, possibly variable, tab stops. **LIST** is a comma-separated list of column positions. A single value (**-x4**) behaves exactly like **--tab-width** (every-N stops). Multiple values pin those stops and repeat the **last interval** past the final entry — e.g. **-x4,8,16** stops at columns 4, 8, 16, 24, 32, ... (the trailing interval of 8 repeats). Overrides **--tab-width**. Mirrors **less -x**.
- **--hex** — render the source as an **xxd**-style hex dump. Mutually exclusive with **--filter**, **--grep**, **--prettify**, **--format**, **--display**, **--record-start**, and **--prompt**.
- **--no-image** — disable image auto-detection; treat image files as raw bytes. Combine with **--hex** for a byte-level view of image data.
- **--blocks** — render detected images in Unicode half-block (■) mode instead of the default character ramp. Each cell encodes two pixel rows.
- **--image-width N** — scale the rendered image to N columns (default: terminal width).
- **--image-protocol auto|kitty|sixel|ascii** — pick the image rendering path (default **auto**). **auto** detects terminal graphics support (Kitty graphics / Sixel) and falls back to ASCII; explicit modes skip detection. See the **Images** section for full behavior.

Source

- **-f**, **--follow** — keep watching the source for new bytes (**tail -f**-style). Jumps to the bottom on startup. Toggle interactively with **Shift-F**. With piped stdin, runs a background reader thread. **For appending writers** (log files); see **--live** for files rewritten in place. On Unix, file rotation (**logrotate**) and truncation are detected on the next poll: the source is re-opened from offset 0, the line index resets, and the status flashes (**F reopened**) for 1 s. After 5 s of silence the indicator becomes (**F idle**) so you can tell the source is being watched but quiet.
- **--follow-suspend-on-motion** — in follow mode, suspend following whenever the user moves (scroll, page, goto-line). Re-engage with **Shift-F**. Matches **less +F** semantics. Default off: today's behavior (movement leaves follow on, auto-scroll just doesn't fire while the viewport isn't at bottom). Bare **G** (goto-bottom) intentionally never suspends — it's the user re-engaging.
- **--follow-name** — accepted for **tail -F** / **less --follow-name** compatibility. No-op: **tess** already re-opens by path on rotation/truncation (since 0.25.0).
- **--exit-follow-on-close** — in follow mode with piped stdin, exit when the upstream writer closes the pipe. Default off (**tess** keeps the captured content visible after EOF). Mirrors **less --exit-follow-on-close**. No-op for file sources.
- **--live** — watch a file for *whole-file* rewrites: when the file's (**mtime**, **size**, **inode**) changes, re-read the entire file, rebuild the line index, and re-render. Use for source files being edited or saved by an AI agent. Different from **--follow** (the two are mutually exclusive). Polling is at the same 250 ms cadence as the rest of the event loop, so saves land within $\frac{1}{4}$ second. Press **R** inside the pager to force an immediate reload. Status line shows (**L**) while active. Caveats:
 - Best for source-file-sized inputs — the whole file is re-read on each change.

- Atomic-rename writers (`vim` , `code` , most editors) work cleanly. Truncate-and-stream writers may briefly flicker if you catch them mid-write; saves typically settle on the next tick.
- Scroll position is preserved (clamped to the new total). If you were at the very bottom, the viewport snaps to the new bottom.
- Requires a file path; not supported on stdin.
- `--tail N` — show only the last `N` logical lines. For files, reverse-scans for the byte offset and only indexes from there forward, so a 10 GB log stays cheap. Mutually exclusive with `--head` . Streaming stdin (`-f` without a file) is not supported. Re-applied on every reload under `--live` .
- `--head N` — cap the visible content to the first `N` logical lines. Mutually exclusive with `--tail` . Re-applied on every reload under `--live` .
- `--encoding LABEL` — decode the input as `LABEL` (a WHATWG charset label) instead of UTF-8. See Character encoding.
- `--diff` — open a side-by-side aligned diff of two files (implies a split). Interactive only. See Diff mode. Runtime: `:diff` .
- `--diff-force` — allow `--diff` past the size cap (large files; may be slow / memory-heavy). Runtime: `:diff!` .
- `--gitdiff` — aligned diff of one file against git: `HEAD` (or a given revision / the `--staged` index) vs the working tree. Requires a git repo. See Git diff.
- `--staged` / `--cached` — with `--gitdiff` , use the staged (index) version as the right side.
- `--diff-ignore-whitespace` — in `--diff` , treat lines differing only in whitespace as equal. Runtime: `:diffws` .

Character encoding

By default `tess` reads input as UTF-8 (invalid bytes show as `<HH>`). For files in a legacy or non-UTF-8 charset, `--encoding LABEL` decodes them so they display as text:

```
tess --encoding iso-8859-1 legacy.log      # café, not caf<E9>
tess --encoding windows-1252 report.txt
tess --encoding shift_jis access.log
```

`LABEL` is any WHATWG encoding label (`utf-8` , `iso-8859-1` , `latin1` , `windows-1252` , `shift_jis` , `euc-jp` , `gbk` , ...), decoded via the `encoding_rs` library. An unknown label is a startup error.

- **Decoding is everywhere.** Rendering *and* matching operate on the decoded text — search (`/`), `--grep` , `--filter` , and `--format` all match what you see, so `/café` finds a Latin-1 `café` on screen.
- **Runtime switch:** `:encoding LABEL` re-decodes the view live; `:encoding` with no argument shows the current label. The setting is global (both panes of a split share it).
- **BOM:** with the default encoding, a leading UTF-8 BOM resolves to UTF-8; an explicit `--encoding` always wins over a BOM.
- **Copy/export emit UTF-8:** `:yank` , `--to-clipboard` , and `-o` / `--stdout` write the decoded text as correct Unicode regardless of the source encoding.

- **--r** and **--hex** are unaffected — raw passthrough emits source bytes verbatim (the terminal decodes), and hex always shows raw byte values.

Two specifics: the label `iso-8859-1` (and `latin1`) uses the **windows-1252** decoder per the WHATWG standard — the right choice for real-world Latin-1 text (they differ only in the `0x80–0x9F` range). And **UTF-16 is not supported** (`--encoding utf-16le / utf-16be` and a UTF-16 BOM are rejected with a message): `tess` splits lines on a lone `0x0A` byte, which UTF-16 code units embed, so a UTF-16 file would misalign after the first line.

Structured logs

- **--format NAME** — parse each line via a named log format (built-in or user-defined). Required by `--filter`.
- **--filter FIELD<op>VALUE** — keep only lines whose parsed `FIELD` matches the predicate. Repeatable; multiple filters AND together. Operators:
 - `=` — exact match. `--filter status=500`
 - `!=` — exact non-match. `--filter status!=200`
 - `~` — regex match. `--filter ip~^10\.`
 - `!~` — regex non-match. `--filter agent!~bot`
 - `<`, `<=`, `>`, `>=` — comparison. Numeric if both the captured value and the predicate value parse as numbers (`f64`), otherwise lexicographic byte order.
`--filter 'status>=500'`, `--filter 'hour>=10'` `--filter 'hour<=12'`,
`--filter 'level>warn'` (lex). When the captured value is non-numeric (e.g. CLF's `-` for “missing size”), comparison falls back to lex and effectively rejects.

> **Shell quoting** (important): in interactive `bash` and `zsh`, the `!` in `!=` / `!~` triggers history expansion (`bash: !=200: event not found`), and the `<` / `>` in `<`, `<=`, `>`, `>=` are treated as input/output redirection. Quote the filter argument with **single quotes** to disable both: > >
`sh \> tess --format apache-combined --filter 'status!=200' access.log \> tess --format apache-combined --filter 'status\>=500' access.log \> tess --format app --filter 'level!~notice' app.log \>` > >

Single quotes are sufficient and prevent any other shell metacharacter (`\`, `$`, etc.) from being interpreted in your regex too. Inside scripts, history expansion is off by default; quoting is only needed at an interactive prompt. Alternatively `set +H` in `bash` disables history expansion for the session.

- **--grep PATTERN** — filter visible lines by regex against the raw line. Repeatable; multiple `--grep` arguments AND. Works on any input — no `--format` required. Composes with `--filter` (both must match) and with `--dim` (non-matches stay visible but faded).

```
tess --grep error access.log
tess --grep error --grep '^\[ ' access.log      # both must match
tess --grep error --dim access.log              # dim non-matches
tess --format apache-combined --filter status=500 --grep timeout access.log
```

- **--or-group NAME** — declare the start of a named OR-group. All subsequent `--or-filter` and `--or-grep` arguments join this group until the next `--or-group` marker. Conditions that appear before any `--or-group` marker form the implicit default group.
- **--or-filter FIELD<op>VALUE** — add a condition to the current OR-group. Accepts the same operators as `--filter`. Requires `--format`. Multiple `--or-filter` values within the same OR-group are OR'd; at least one must match. Across OR-groups the constraint is

AND: every non-empty OR-group must have ≥ 1 match. `--filter` / `--grep` predicates (required matches) are AND'd with all OR-group constraints.

- `--or-grep PATTERN` — add a raw-line regex condition to the current OR-group. Works on any input — no `--format` required. Shares the same OR/AND group semantics as `--or-filter`. A record is shown when: ALL `--filter` / `--grep` conditions match AND every non-empty OR-group has at least one matching condition.

```
# ssh-service lines matching any brute-force signature:
tess --format ssh --filter service=ssh \
    --or-grep 'failed password' --or-grep 'invalid user' access.log

# Two independent OR-groups (both must have a hit):
tess --format app --filter 'level=ERROR' \
    --or-grep 'timeout' --or-grep 'deadline' \
    --or-group http --or-filter 'status>=500' --or-filter 'status=429' app.log
```

The status line shows `[or]` whenever any OR-group is active. The `<or-tag>` placeholder is also available in `--prompt` templates.

- `--dim` — render non-matching lines visibly faded instead of hiding them. Works with `--filter`, `--grep`, or both. Keeps surrounding context visible.
- `--display TEMPLATE` — reformat each parsed line into a custom view. Placeholders `<fieldname>` are replaced with the captured value (empty if the regex didn't capture the field on this line). `\<` is a literal `<`, `\\` is a literal `\` other `\X` is left as-is. Lines that don't parse against the format regex fall back to their raw form so no data is silently dropped. Requires `--format`. Overrides the format's `display` key (if set in `formats.toml`). Affects both the interactive view and `--output` / `--stdout`. Search runs against the rendered template (so what you see is what you can find); filtering still operates on the raw captures. Mutually exclusive with `--prettify`.
- `--list-formats` — print available formats and their named fields, then exit.

Runtime filtering & per-pane predicates

Predicates can also be set live, from the colon prompt — they apply to the **focused pane**, so in a split you `Tab` to a pane and give it its own:

Command	Effect
<code>:grep PAT / :nogrep</code>	Set / clear a raw-regex filter (no format needed).
<code>:filter FIELD<op>VALUE / :nofilter</code>	Set / clear a field filter. Needs a format on the pane (else a hint flashes). Sets a single spec (replaces).
<code>:format NAME / :noformat</code>	Set / clear the pane's format (enables <code>:filter</code> / <code>:display</code>). <code>:noformat</code> also clears the filter.
<code>:display TEMPLATE / :nodisplay</code>	Set / clear the display template (needs a format).

Compile errors (bad regex, unknown format) flash on the status line and leave the view unchanged. These work single-pane too. Case sensitivity follows the pane's `:case` setting.

For the split view, the second pane can be seeded at startup with its own predicates, mirroring the main flags:

- **--right-grep PATTERN** (repeatable, AND'd) — pane B raw-regex filter.
- **--right-filter FIELD<op>VALUE** (repeatable, AND'd) — pane B field filter; requires `--right-format`.
- **--right-format NAME** — pane B parse format.
- **--right-display TEMPLATE** — pane B display template.
- **--right-ignore-case** — pane B smart-case (analog of `-i`); mutually exclusive with `--right-IGNORE-CASE`.
- **--right-IGNORE-CASE** — pane B force case-insensitive (analog of `-I`).

Pane B's case mode is **independent** of the global `-i` / `-I` (defaults to case-sensitive); the `--right-ignore-case` / `--right-IGNORE-CASE` flags set it, and it governs both pane B's `--right-*` predicates and its interactive search. Per-pane case can also be changed live with `:case` on the focused pane.

```
tess --split access.log access.log \
  --format nginx-combined --filter status=500 \
  --right-format nginx-combined --right-filter status=404
# left pane: 500s   right pane: 404s
```

The `--right-*` flags are independent of the left (no inheritance) and only meaningful with `--split` / `--diff`. (OR-groups and `--dim` remain global / startup-only; per-pane predicates are inactive while `:diff` is on.)

Pretty-printing

- **--prettify** — reformat the file's content for human reading. Supports **JSON, YAML, TOML, XML, HTML, CSV**. Type is detected from the filename extension (`.json`, `.yaml` / `.yml`, `.toml`, `.xml`, `.html` / `.htm`, `.csv`) and falls back to a quick byte sniff for unextended files. **Static files only** — not allowed with `--follow`, `--live`, or `--filter` (which would all conflict with reshaping the byte stream). Layout only — no syntax highlighting, so search and `--filter` (when used separately) keep working byte-cleanly.
- **--content-type NAME** — override detection. Values: `auto` (default — same as not passing this flag), `raw` (force prettify off, even if `--prettify` is also given), `json`, `yaml` (alias `yml`), `toml`, `xml`, `html` (alias `htm`), `csv`. Setting this implies `--prettify` unless the value is `auto` or `raw`.

If a transform fails to parse, `tess` falls back to showing the raw content and the status line shows `[pretty:<type>:err]` so you know why nothing changed.

CSV cells are aligned into a fixed-width table; cells longer than 60 characters are truncated with an ellipsis (...) so a single runaway free-text column doesn't blow up the layout.

Batch (non-interactive) output

- **-o FILE**, **--output FILE** — apply `--filter` / `--head` / `--tail` / `--prettify` to the source, write the surviving logical lines as **raw bytes** (one per line, separated by `\n`) to `FILE`, and exit. Use `FILE = -` to write to stdout. The terminal alt-screen and raw mode are not entered, so this is safe to run from scripts and CI.
- **--stdout** — synonym for `-o -`.
- With `--follow`, the run doesn't exit after the initial pass: it keeps polling the source and appending matching new lines as they arrive (`Ctrl-C` cleanly closes the file). Useful for `tess -f --filter status~^5 -o errors.log` to harvest only error lines from a live log.

- Incompatible with `--live` (which is a “watch a file rewrite, render the new view” feature — there’s no view in batch).
- `--dim`, `-N` (line numbers), and `-S` (chop) are viewport-only concerns and are silently ignored in batch mode. The output is always the raw bytes of matching lines, exactly as they appear in the source (or in the prettified stream when `--prettify` is on), so the file stays `grep-/awk-/diff-able`.

Clipboard

tess can read from and write to the system clipboard, and yank lines interactively. This is a tess-native feature (not a `less` flag). It shells out to the OS clipboard tools — `pbcopy` / `pbpaste` on macOS; on Linux it tries `wl-copy` / `wl-paste` (Wayland) first, then `xclip`, then `xsel`. If the chosen tool is missing or fails, the error is surfaced on the status line.

- `--from-clipboard` — use the system clipboard contents as the input source instead of a file or stdin. Conflicts with file arguments, and is rejected together with `--follow` / `--live` (the clipboard isn’t a growing/rewritten source).
- `--to-clipboard` — a batch sink, analogous to `-o` / `--stdout`: apply any `--filter` / `--grep` / `--head` / `--tail` / `--prettify`, copy the resulting lines to the clipboard, and exit without entering the pager. Conflicts with `-o` / `--stdout`, and is rejected with `--follow`.
- `--clipboard` — enable interactive yank. With it on, the `:yank` colon command copies the current line to the clipboard. There is also a remappable `clipboard-yank-line` keybinding for this, left **unbound by default** so it doesn’t clobber `y` (scroll-up) — bind it yourself in `~/.config/tess/keys.toml` if you want a single-key yank.

```
echo "hello" | pbcopy && tess --from-clipboard          # view clipboard
tess --format apache-combined --filter status=500 \    # filtered → clipboard
  --to-clipboard access.log                            # then :yank a line
tess --clipboard app.log
```

Other

- `-h`, `--help` — print a flag list (sorted alphabetically by long name) and exit.
- `--manual` — print this manual to stdout and exit. Pipe to a pager if you want to scroll: `tess --manual | less`.
- `--examples` — print a short, curated list of practical usage recipes and exit. Lighter than `--manual`.
- `--mouse` — explicit-on alias for mouse capture (kept for compatibility; capture is **on by default** since 0.56.0). Conflicts with `--no-mouse`.
- `--no-mouse` — disable mouse capture at startup, restoring native terminal text-selection. Conflicts with `--mouse`. See Mouse capture for the full story and runtime controls.
- `--wheel-lines N` — the absolute number of body lines scrolled per mouse-wheel notch. Default 3.
- `-R`, `--RAW-CONTROL-CHARS` — accepted alias for tess’s default ANSI-interpret mode. A no-op, provided for drop-in `less -R` muscle memory. Conflicts with `-r` (true raw passthrough) and `--no-color`.
- `-#`, `--shift N` — column count for the `</>` horizontal-scroll commands. 0 (the default) keeps the half-screen behavior. Mirrors `less -#` / `--shift`.
- `--split` — open a side-by-side vertical split of the first two file arguments (or two views of a single file). Interactive only. See Split view. Runtime equivalent: `:vsplit`.

- **--hsplit** — like `--split` but **horizontal** (stacked rows). Runtime: `:hsplit` flip an open split with `:rotate`. See Horizontal (stacked) split.
- **--scroll-lock** — start a `--split` session with the two panes scroll-locked together. Only meaningful with `--split` ignored otherwise. See Synchronized scrolling. Runtime toggle: `= / :scrolllock`.
- **--incsearch** — enable incremental search (off by default). See Search. Toggle at runtime with `:incsearch`. Mirrors `less --incsearch`.
- **--prompt TEMPLATE** — override the built-in status line with a custom template. Placeholders `<field>` expand to live values (see Customizing the status line). CLI `--prompt` overrides any `prompt` key in the active format. Not allowed with `--hex`.
- **-V, --version** — print version.

Interactive keys

Key(s)	Action
<code>↓ j e Ctrl-E Return</code>	Scroll down 1 screen line (walks through wrap rows of long lines)
<code>↑ k y Ctrl-Y</code>	Scroll up 1 screen line
<code>J</code>	Jump to start of next <i>logical</i> line, skipping any remaining wrap rows
<code>K</code>	Jump to start of current/previous <i>logical</i> line
<code>Space f Ctrl-F PgDn</code>	Page down
<code>b Ctrl-B PgUp</code>	Page up
<code>d Ctrl-D</code>	Half-page down
<code>u Ctrl-U</code>	Half-page up
<code>g < Home</code>	Go to top
<code>G > End</code>	Go to bottom
<code>/ pattern Enter</code>	Forward regex search; <code>Esc</code> cancels the prompt
<code>? pattern Enter</code>	Backward regex search
<code>n</code>	Repeat last search (same direction)
<code>N</code>	Repeat last search (opposite direction)
<code>-N (dash, then N)</code>	Toggle line numbers
<code>← →</code>	Scroll left / right by half the screen width (chop mode and image view only; no-op in wrap, hex, or raw)
<code>Shift-← Shift-→</code>	Scroll left / right by 8 columns
<code>Shift +scroll / horizontal trackpad scroll</code>	Scroll left / right by 8 columns (requires mouse capture on; <code>Shift +wheel</code> works everywhere, native horizontal swipe only where the terminal reports it)
<code>-S (dash, then S)</code>	Toggle chop / wrap
<code>Shift-F</code>	Toggle follow mode

Shift-P	Toggle pretty-print on/off (only when <code>--prettify</code> was active at startup)
p	Animated image: pause / resume (revives a finished animation); no-op without an animation
. ,	Animated image: step one frame forward / back (auto-pauses)
Backspace	Animated image: restart from the first frame
Tab / BackTab	Cycle the focused pane forward / backward in a split (see Split view no-op without a split)
Ctrl-X z	Toggle zoom (maximize) the focused pane in a split
=	Toggle synchronized scrolling in a split (see Synchronized scrolling flashes a hint without a split)
]c [c	In diff mode, jump to the next / previous change hunk (see Diff mode no-op outside diff)
r Ctrl-L	Force redraw
Shift-R	Force-reload from disk (with <code>--live</code> no-op otherwise)
F1	Open the help overlay (also <code>:help</code> / <code>:h</code> at the colon prompt)
q Q Ctrl-C	Quit

In hide-mode filtering, scroll/page/goto operate on visible (matching) lines — the viewport skips past hidden ones.

Mouse capture

Mouse capture is **on by default** since 0.56.0: the scrollwheel scrolls the body, rows in the file picker / help overlay are clickable, and (in a split) a left-click focuses the pane under the cursor — all without any extra flag.

Text-selection trade-off. Most terminals disable their native text selection while a program holds mouse capture. To select and copy text with the mouse, hold **Shift** while clicking/dragging (on iTerm2 / macOS, hold **Option** instead). Alternatively, disable capture entirely with `--no-mouse` or `:mouse off`.

Startup flags:

- `--no-mouse` — disable capture at startup (preserves native text selection). Conflicts with `--mouse`.
- `--mouse` — explicit-on alias (kept for compatibility; a no-op given the default). Conflicts with `--no-mouse`.

Runtime toggle (`:mouse`):

Command	Effect
---------	--------

:mouse	Flip the current capture state
:mouse on	Enable capture
:mouse off	Disable capture

The `mouse-toggle` keybinding name is available for `~/.config/tess/keys.toml` (unbound by default — add it there to get a one-key toggle).

Persistent default (`formats.toml`): set `mouse = false` in a `[settings]` table to make capture off the default for every invocation (CLI flags win over it):

```
# ~/.config/tess/formats.toml
[settings]
mouse = false
```

Status indicators: while capture is off, the status line shows a `[nomouse]` badge. The `<mouse>` prompt placeholder expands to `nomouse` when off, empty when on — useful for custom `--prompt` templates.

Horizontal scrolling

Horizontal scrolling is active in **chop mode** (`-S`) and **image view** (`--image-width N` wider than the terminal). It is a no-op in wrap mode, hex (`--hex`), and raw (`-r`) mode.

- `← / →` — scroll left / right by half the screen width.
- `Shift-← / Shift-→` — scroll left / right by 8 columns.
- Mouse (requires capture on; see Mouse capture) — 8-column step, via either a native horizontal swipe (`ScrollLeft / ScrollRight`) or `Shift + scroll-wheel`, **whichever your terminal reports**. This is terminal-dependent: iTerm2, kitty, WezTerm, and xterm report one or both; **Warp and macOS Terminal.app report neither** — they deliver only bare vertical scroll with no horizontal variant and no scroll modifiers, so there is no gesture tess can distinguish. In those terminals, use the keyboard (`← / →`, `Shift-← / Shift-→`) to scroll horizontally.

All four bindings are remappable in `~/.config/tess/keys.toml` via the command names `hscroll-left`, `hscroll-right`, `hscroll-left-step`, `hscroll-right-step`. Scrolling fully left clamps back to column 0.

The `← / →` step defaults to half the screen width, but `-# / --shift N` overrides it with a fixed column count (`0` keeps the half-screen default). The `Shift-← / Shift-→` 8-column step is unaffected.

The **line-number gutter** (`-N`) stays fixed while text scrolls. For chopped text, a `<` marker appears at the left edge when content extends further left (analogous to the existing `> / --rscroll` right-edge marker). Images shift cleanly with no edge markers. When scrolled past column 0, the status line shows a `»{col}` offset readout; `<col-offset>` is available as a `--prompt` template placeholder.

Frozen left content-columns (`--header ,C`): in chop mode the first `C` display columns stay pinned while `← / →` scroll the rest of each line, with a dim `|` divider between the frozen region and the scrolled remainder. The freeze engages only once scrolled; the dedicated `<` left-edge marker gives way to the divider. A width-2 character straddling the boundary is

dropped at the edge. This also applies per pane in the split view (each pane honors its own `--header`).

Split view

Open multiple panes side by side in vertical columns for lightweight compare within the pager.

```
tess --split app.log app.log.1      # two files, side by side
tess --split a.log b.log c.log      # N panes — one per file
tess --split big.log                # one file → a second independent view
```

`--split` opens **one pane per file argument** (≥ 2 files $\rightarrow$ that many panes; a single file $\rightarrow$ 2 views of it).

Per-pane flags: the `--` form

When you want each pane to carry **its own** flags, separate the panes with a standalone `--`. Each `--`-delimited section is a full per-pane view spec — a file plus its per-view flags:

```
tess a.log --grep ERROR -- b.log --grep WARN -- c.log
tess access.log --format apache-common --filter status=500 -- access.log
```

Rules:

- The **first** section (before any `--`) carries the **session globals** (`--mouse`, `exit/ -X` modes, theming, keybindings, ...) plus the focused pane. Later sections are **view-only**: their global flags are ignored.
- Per-view flags honored per section: the file, `--grep`, `--filter`, `--format`, `--display`, `--encoding`, `-i/ -I`, and the display flags (`-N`, `-S`, `--wordwrap`, `--tabs`, `--header`, `--hex`, ...). User-defined groups (`--<groupname>`) expand per-section.
- `--diff a -- b` (exactly two sections) renders the **aligned diff**.
- The `--` form is **mutually exclusive** with `--split/ --right-*` (using both is an error). OR-groups (`--or-*`) and `+CMD` are **first-section-only** (an `--or-*` in a later section is rejected; put them before the first `--`).
- The first pane must name **at most one** file (stdin or a single path).
- A `--` that would leave an **empty** section keeps its POSIX *end of options* meaning rather than splitting — so `tess -- -dash-named-file` opens a file whose name starts with `-`, and a trailing `tess a --` is harmless.

`--split` is **interactive only** (ignored for `--to-clipboard/ --stdout` batch runs). At runtime:

- `:vsplit [file] / :split [file]` — open a split. With a path argument it opens that file in the new pane (a leading `~/` is expanded); with no argument it duplicates the focused file at its current scroll position. Duplicating stdin isn't possible (there's no path to reopen) — that case reports an error on the status line.
- `:only / :close` — collapse back to a single pane (the focused one).

Each pane is a full viewport with its **own** scroll position, search, marks, and follow/tail state.

`Tab` cycles the focused pane **forward**, `BackTab` (Shift-Tab) cycles **backward** — scroll, search, and colon commands target the focused pane, while the others keep following / tailing on their own (including log-rotation re-open). The keys are remappable in

`~/.config/tess/keys.toml` via `focus-other-pane / focus-prev-pane`. With mouse capture on (the default), a **left-click in a pane** focuses it directly — handy with three or more panes where cycling is slower. Clicking works under scroll-lock too (only focus moves), is a no-op in diff mode (focus is locked there), and clicking the already-focused pane does nothing.

Zoom (maximize). `Ctrl-X z` (or `:zoom`, or a `zoom-pane` binding) toggles the focused pane between its split size and full-screen — like `tmux`'s pane zoom. The other panes are hidden (still following/tailing), not closed, and a `[zoom]` badge marks the state. Switching focus or running a structural command (`:vsplit / :hsplit / :only / :layout / :rotate`) or `:diff` unzooms first. It is a no-op in a single pane or in diff mode. `Ctrl+Shift+X` is not bound by default (terminals can't reliably distinguish it from `Ctrl+X`); bind

`ctrl-shift-x = "zoom-pane"` in `~/.config/tess/keys.toml` if your terminal supports the Kitty/CSI-u keyboard protocol.

Panes are separated by vertical dividers, and each gets its own status segment (`1/N` width); the focused pane's is prefixed with `*`. Each pane needs at least 8 usable columns: in a terminal too narrow to fit them all, the focused pane renders full-width until the window is widened.

Under `--mouse`, the wheel (vertical, `Shift+wheel`, and native horizontal) scrolls the pane the **cursor is over** — not necessarily the focused one — and does not change focus. (Exceptions: while scroll-locked the panes move together, and in diff mode the aligned pair scrolls as one view.)

Horizontal (stacked) split

By default the split is **vertical** (side-by-side columns). A **horizontal** split stacks the panes as rows instead, each getting a height slice with its **own status row** at the bottom of its slice (the status rows double as the separators — there is no extra divider row).

```
tess --hsplit app.log app.log.1 # stacked rows at startup
tess --hsplit big.log           # one file → two stacked views
```

At runtime:

- `:hsplit [file] / :hsp` — open a stacked split (no argument duplicates the focused file at its scroll position, like `:vsplit`).
- `:rotate` — flip the **current** split's orientation (vertical ↔ horizontal) in place, without reopening; panes, sources, and scroll positions are kept. No-op (with a flash) when there's a single pane or in diff mode.

Orientation is **whole-split** — every pane shares it; you can't mix columns and rows in one view (nested grids are a future addition). `--split / :vsplit` stay vertical, and a `--`-form split opens vertical but can be `:rotate`d. Each horizontal pane needs at least **2 rows** (1 body + 1 status); in a terminal too short to fit them all, the focused pane renders full-screen until there's room — the height analog of the vertical ≥ 8 -columns rule. Under `--mouse`, the wheel targets the pane the cursor's **row** is over. Diff mode stays vertical (`--hsplit --diff` is rejected; `:rotate` is a no-op while diffing).

Synchronized scrolling (`:scrolllock`)

By default the two panes scroll independently. Press `=` (or run `:scrolllock`, or start with `--scroll-lock`) to **lock** them together: scrolling the focused pane scrolls the other by the same number of logical lines. The focused pane's status shows `[lock]`.

The lock is **relative** — the offset between the two panes' top lines is captured the moment you enable it. So scroll each pane to the region you want beside the other *first*, then press `=` to freeze that alignment and scroll together. Coupling is by logical line, so the panes stay aligned by content even if they wrap differently. The partner is re-derived from the fixed offset on every move, so scrolling one pane to its top/bottom and back **restores** the alignment exactly — no drift. `Tab` swaps focus without moving either pane.

While locked, the **focused pane drives**: only it auto-tails under `--follow` / `--live` the other pane's position follows the lock. Press `=` again to unlock (the panes stay where they are and become independent); `:only` also clears the lock. `=` is remappable as `scroll-lock-toggle` in `~/.config/tess/keys.toml`, and `<lock>` is a `--prompt` placeholder (`lock` when on, empty when off).

v1 limitations. The split is **vertical only** (columns) — no horizontal (stacked) split. Aligned `diff` (`:diff`) requires **exactly 2 panes**. Per-pane predicates: `--filter` / `--grep` / `--format` / `--display` apply to the first pane, and `--right-*` to the second; panes 3+ use the shared flags (a uniform per-pane mechanism is planned). Because the split compositor works on rendered cells, **protocol images** (Kitty/Sixel) render as **ASCII** and `-r` **raw** content renders through the cell pipeline (Interpret) while split. Runtime `:vsplit` panes also do not apply `--tabs`, `--header`, or status-prompt theming (the startup `--split` pane does). Frozen left columns (`--header`, `C`) therefore apply per pane for startup-configured panes (each pane honors its own `--header`), but not for runtime `:vsplit` panes (which don't apply `--header`). See `OUT-OF-SCOPE.md`.

Diff mode (`--diff` / `:diff`)

Diff mode aligns the two panes by a line-level diff so matching lines sit beside each other — a side-by-side compare inside the pager.

```
tess --diff old.conf new.conf      # open the split and align in one shot
```

At runtime, on an existing split: `:diff` enters diff mode, `:diff!` bypasses the size cap (below), `:nodiff` returns to the plain split.

- **Alignment + fillers.** A Myers diff classifies each line pair as *equal*, *changed*, *added*, or *removed*. Inserted/deleted lines show a blank **filler** on the opposite side so the rest stays aligned. Gutter signs `/ ~ / + / -` and colors mark each line; **changed** lines additionally get **intra-line highlighting** of the differing characters. Long lines wrap with pair-padding (each aligned pair occupies as many rows as its taller side).
- **Navigate changes:** `]c` / `[c` jump to the next / previous change hunk (skipping equal regions); the status shows `[diff i/n]`. Remappable as `diff-next-change` / `diff-prev-change`.
- **Ignore whitespace:** `--diff-ignore-whitespace`, or toggle live with `:diffws`, treats lines differing only in whitespace as equal.
- **Charset-aware:** diff honors the active `--encoding` / `:encoding` — the alignment display and the intra-line char diff both run on the decoded text.

- **Huge files:** diffing loads both files fully and runs an $O(ND)$ diff, so if either exceeds 500k lines `tess` refuses with a message and stays in the plain (sync-scrollable) split — unless you force it with `:diff!` / `--diff-force`. Diff is a **snapshot**: `--follow` / `--live` auto-update is suspended while it's active.

Diff-mode v1 limits. Diff operates on **raw file lines** (no `--filter` / `--format` / `--display`). Intra-line highlighting is computed only for changed lines that fit a single row on their pane. `Tab` (focus swap) is locked in diff mode — the panes scroll as one. A numbered `<n>G` jumps to the top (use `]c` / `[c` to move between changes).

Git diff (`--gitdiff`)

```
tess --gitdiff src/foo.rs           # HEAD vs working tree
tess --gitdiff HEAD~3 src/foo.rs    # an older commit vs working tree
tess --gitdiff v1.0 v2.0 src/foo.rs # commit vs commit
tess --gitdiff --staged src/foo.rs   # HEAD vs the staged (index) version
tess --gitdiff HEAD~1 --staged foo  # an older commit vs the index
```

`--gitdiff` is the same aligned side-by-side diff as `--diff`, but it sources its sides from git instead of a second file. It's the quick “what have I changed” view, with changes highlighted exactly as in `--diff`.

Forms. Revisions are **leading positionals** — the **last positional is always the file**, the 0, 1, or 2 before it are revisions:

Invocation	left (old)	right (new)
<code>--gitdiff FILE</code>	HEAD	working tree
<code>--gitdiff REV FILE</code>	REV	working tree
<code>--gitdiff R1 R2 FILE</code>	R1	R2
<code>--gitdiff --staged FILE</code>	HEAD	index
<code>--gitdiff REV --staged FILE</code>	REV	index

`--staged` has the alias `--cached`. A `REV` is anything `git` resolves (`HEAD~3`, a branch, tag, or SHA).

- A side where the file **doesn't exist** in that revision/index (or, for the working tree, on disk) shows as empty — all-added or all-removed.
- Honors `--diff-ignore-whitespace` and `--diff-force` all the diff navigation (`]c` / `[c`, `:diffws`) applies.
- Requires a **git repository** and **one file**. Errors cleanly on: not a git repo, no commits yet (unborn `HEAD`), a bad/typo'd revision (bad revision '`\<rev\>`'), `--staged` with two revisions, `--staged` without `--gitdiff`, more than three positionals, or combining with `--diff` / `--split` / `--right-*` / the `--` per-pane form.

Still deferred: multi-file diffs and rename detection. Range syntax as one token (`R1..R2`) isn't supported — pass two positionals instead.

Search

Pressing `/` opens a search prompt at the bottom of the screen. Type a regex (the same flavor as `--filter` uses), then `Enter` to execute or `Esc` to cancel. `?` does the same backward. The matched logical line scrolls to the top of the viewport, and within every

visible row the matched **phrase** itself is rendered in reverse-video (not the whole row). `n` repeats the last search in its original direction; `N` repeats it the other way. Pressing `/` (or `?`) followed by `Enter` with an empty pattern repeats the last search in the typed direction (just like `n` / `N`). Search wraps at the end of the source.

When a filter is active, search interacts with it predictably: in hide mode, only currently-visible (matching) lines are searched. In dim mode, lines stay dimmed but the matched phrase within each line is still highlighted so it pops out of the surrounding context.

The status line picks up `[/<pattern>]` (or `[?<pattern>]`) while a search is set.

Incremental search (`--incsearch`)

With `--incsearch`, the search prompt previews as you type: after each keystroke the view jumps to and highlights the first match found from where the prompt opened. `Enter` commits to the previewed match; `Esc` restores the original position (the preview is abandoned). Off by default; mirrors `less --incsearch`. Toggle it at runtime with `:incsearch` (no arg flips it; the change takes effect on the next search prompt).

Case sensitivity

Three case policies control how `/`, `?`, `--grep`, and `--filter`'s regex operators (`~` / `!~`) match input:

Mode	Behavior
Sensitive (default)	Pattern matches case-exactly.
Smart (<code>-i</code> , <code>--ignore-case</code>)	Case-insensitive UNLESS the pattern contains an uppercase character. Matches <code>less / ripgrep / vim smartcase</code> .
Insensitive (<code>-I</code> , <code>--IGNORE-CASE</code>)	Always case-insensitive regardless of pattern.

Runtime cycling via `:case` (no arg) or `:case sensitive|smart|insensitive`. The active search re-compiles automatically when the mode changes, so the result set updates without retyping the pattern.

Suppressing match highlights

`-G / --no-hilite-search` starts up with search-match highlights disabled — searches still navigate (`n` / `N` jump to matches), but the visual reverse-video on the matched phrase is suppressed. Toggle at runtime with `:hlsearch` / `:nohlsearch`.

`-g / --hilite-search` narrows highlighting the other way: only the match you last jumped to is highlighted (the landed line's first match per display row), rather than every occurrence of the pattern. `-G` (no highlight at all) takes precedence over `-g`. Mirrors `less -g`.

Skipping the screen on search (`-a`)

`-a / --search-skip-screen` makes a forward search (and `n`) start *below* the last displayed line instead of just after the top line — so matches that are already visible on the current screen are skipped, jumping straight to the next match below. Backward search is unaffected (it already starts above the screen). Works in line, filter/hide, and records modes, and per split pane. Toggle at runtime with `:search-skip-screen`. Mirrors `less -a`.

Tildes past end-of-file (--tilde)

By default `tess` shows blank lines below the end of a short file. `--tilde` shows a dim `~` on those rows instead — the classic `less` appearance. Toggle at runtime with `:tilde`. Note the direction differs from `less`: `less` shows tildes by default and `--` disables them, whereas `tess` defaults to blank and `--tilde` enables them (a deliberate divergence; long flag only, no `--`).

Startup commands (+CMD)

Like `less +CMD` and `vim +CMD`, `tess` accepts startup commands as argv tokens beginning with `+`. They're applied against the viewport just before the event loop spins up. Multiple `+CMD`s are allowed and apply in argv order.

Form	Effect
<code>+G</code>	Jump to bottom of file.
<code>+NUM</code>	Jump to 1-indexed line NUM.
<code>+/pat</code>	Forward-search for <code>pat</code> jump to first match. Honors <code>-i</code> / <code>-I</code> .
<code>+?pat</code>	Backward-search; jump to first prior match.

```
tess +/error access.log      # open at first 'error' match
tess +500 huge.csv           # open at line 500
tess +G app.log              # open at end (like '-f' without the follow)
```

`-p PATTERN` / `--pattern PATTERN` is the flag form of `+/PATTERN`: open scrolled to the first line matching `PATTERN` (honoring `-i` / `-I`). If both a `+/` command and `-p` are given, `-p` is applied last. Mirrors `less -p`.

Exit-on-EOF and one-screen pager

Flag	Behavior
<code>-e</code> / <code>--quit-at-eof</code>	Quit on the <i>second</i> forward scroll/page that lands at EOF. Mirrors <code>less -e</code> .
<code>-E</code> / <code>--QUIT-AT-EOF</code>	Quit the <i>first</i> time EOF is reached. Mirrors <code>less -E</code> .
<code>-F</code> / <code>--quit-if-one-screen</code>	If the entire source fits on one screen, print verbatim and exit — no pager.
<code>-K</code> / <code>--quit-on-intr</code>	No-op; Ctrl-C already quits. Accepted for compatibility.

Display tweaks

Flag	Behavior
<code>-s</code> / <code>--squeeze-blank-lines</code>	Collapse runs of blank lines to a single blank at display time. Line counts / search / tag-jumps unaffected.
<code>--header=L[,C]</code>	Pin top L source rows (and the left C columns, when horizontal scroll lands). Long pinned lines truncate at the right edge. Runtime: <code>:header L [C]</code> .
<code>--rscroll=CHAR</code>	Character at the right edge when a line is chopped in <code>-S</code> mode. Default <code>></code> . Pass <code>''</code> to disable.

<code>-z N / --window=N</code>	PageDown / PageUp step size in lines (default: full body height). Half-page commands always advance by half the screen regardless.
<code>--wordwrap</code>	In wrap mode, break on whitespace boundaries instead of mid-character. Falls back to mid-character break when no whitespace fits.

Skipping the alt-screen (-X)

`-X / --no-init` prevents `tess` from switching to the terminal's alternate screen on startup. The body paints to the primary screen and remains in scrollback after exit. Pairs naturally with `-F` for git-pager-style workflows, and is invaluable when piping through multiple commands.

```
git --no-pager diff --color=always | tess -X -F
```

Option-toggle prefix (-)

Borrowed from real `less` : pressing `-` enters a one-shot option-prefix mode. The next keystroke selects which option to flip:

<code>-</code> then...	Effect
<code>N</code>	Toggle line numbers
<code>S</code>	Toggle chop / wrap
<code>F</code>	Toggle follow (also available as <code>Shift-F</code> directly)
<code>P</code>	Enter the pretty-print sub-prefix (see below)

Lowercase variants work too (`-n` , `-s` , `-f` , `-p`). Any other key after `-` cancels the prefix harmlessly.

After `-P` , one more keystroke sets the content type:

<code>-P</code> then...	Effect
<code>j</code>	Force JSON
<code>y</code>	Force YAML
<code>t</code>	Force TOML
<code>x</code>	Force XML
<code>h</code>	Force HTML
<code>c</code>	Force CSV
<code>a</code>	Auto-detect from current bytes
<code>r</code>	Raw (turn prettify off)

Both `-P` letters are case-insensitive. Any other key cancels the sub-prefix.

Status line

The bottom row shows current state. Format:

```
<source> <top>-<bottom>/<total> <pct>% +<wrap>/<wraps> [<format>] [grep]
[filter]/[dim] [/<search>] [pretty:<type>] (L) (F)
```

- **<source>** — file path or (stdin) .
- **<top>-<bottom>/<total>** — currently visible line range over total. With `--filter` (hide mode) this is `top-bottom/<matched>/<total>` .
- **<pct>%** — position percentage.
- **+<wrap>/<wraps>** — only shown when scrolled inside a wrapped line. Tells you which wrap row of the current logical line is at the top of the viewport (e.g. `+12/50` means wrap row 12 of a 50-row line). Lets you see that `j` is making progress through a long line; goes away when you reach the next logical line.
- **[<format>]** — present when `--format` is active (e.g. `[apache-combined]`).
- **[filter] / [dim]** — present when filtering, indicating mode. With `--grep` active, an additional `[grep]` token appears.
- **[/<search>] / [?<search>]** — active search pattern (forward or backward). Cleared only when a new search is set or you exit.
- **[pretty:<type>]** — present when `--pretty` is active. `<type>` is one of `json / yaml / toml / xml / html / csv` . Suffix `:err` indicates the last transform failed to parse; raw content is shown.
- **(L)** — present when `--live` is on. The file is being watched for whole-file rewrites; `R` forces an immediate reload.
- **(F)** — present when follow mode is on. New bytes auto-scroll into view if you're at the bottom.
- **+** suffix on `total` — the source may still grow (streaming stdin, follow mode, or live mode).

While a search prompt is open, the entire status row is replaced with `/<typed-so-far>` (or `?...`). Enter commits, Esc cancels, Backspace edits.

The built-in default status format appends a right-aligned `:help` discoverability hint at the far-right edge of the status line. Custom prompts (`--prompt TEMPLATE` or `prompt = in formats.toml`) do not include this hint — only the built-in default does.

Customizing the status line

Override the built-in status format with a templated string. Same `<field>` syntax as `--display` :

```
tess --prompt '<label> <pct>%' file.log
tess --prompt '<label> <rec-block> <pct>%<grep-tag><hide-tag>' --format app
file.log
```

Set a per-format default in `~/.config/tess/formats.toml` :

```
[format.app]
regex = '^(?P<ts>\S+) (?P<level>\w+) (?P<msg>.+)$'
prompt = '<label> <top>-<bottom>/<total> <pct>%<filter-tag><hide-tag>'
prompt_style = 'fg=bright-blue,bold' # optional — see Theming below
```

CLI `--prompt` overrides `format.prompt`, which overrides the built-in default. The built-in default reproduces `tess`'s standard status format.

CLI `--prompt-style` overrides `format.prompt_style`, which overrides `--status-style`. See [Color and ANSI escapes → Theming](#).

Available placeholders:

Placeholder	Resolves to
<code><label></code>	source label (filename / stdin)
<code><top></code> / <code><bottom></code> / <code><total></code>	visible line range and total
<code><pct></code>	percent through file (0–100)
<code><rec-top></code> / <code><rec-bottom></code> / <code><rec-total></code>	record range and total (records mode only)
<code><rec-block></code>	<code>L<top>-<bot>/<total></code> <code>R<rec-top>-<rec-bot>/<rec-total></code> in records mode; <code><top>-<bot>/<total></code> in line mode
<code><wrap-offset></code>	<code>+N/M</code> indicator when inside a long-wrapped line
<code><format-tag></code>	<code>[<format-name>]</code> when <code>-format</code> is active
<code><filter-tag></code>	<code>[<format-name>]</code> when <code>-filter</code> is active
<code><grep-tag></code>	<code>[grep]</code> when <code>-grep</code> is active
<code><hide-tag></code>	<code>[hide]</code> or <code>[dim]</code> when a predicate is active
<code><search-tag></code>	<code>[/pattern]</code> or <code>[?pattern]</code> while searching
<code><pretty-tag></code>	<code>[pretty:<type>]</code> when prettify is on
<code><live-tag></code> / <code><follow-tag></code>	<code>(L)</code> / <code>(F)</code> markers

Tag placeholders that aren't active resolve to empty strings (with no surrounding whitespace), so a template like `<label><filter-tag>` cleanly collapses when there's no filter.

Escape `\<` for a literal `<` and `\\` for a literal `\`. Unknown placeholders cause a startup error pointing at the offending field name.

Customizing key bindings

Override `tess`'s default keybindings via `~/.config/tess/keys.toml`:

```
[bindings]
"j"      = "scroll-down"      # bind a single character
"shift-j" = "scroll-logical-down" # case-equivalent: "J" also works
"f1"     = "toggle-line-numbers" # function keys
"ctrl-r" = "reload"           # modifiers stack: ctrl-shift-l works too
"f2"     = "!git status"      # `!` prefix runs a shell command
```

Key spec grammar:

- Single character: `"j"`, `"/"`, `"%"`, `"!"`.

- Named special: `esc`, `enter`, `tab`, `backspace`, `space`, `f1 - f12`, `up / down / left / right`, `pgup / pgdn`, `home / end`.
- Modifiers (stackable): `ctrl-`, `alt-`, `shift-`.
- Bare uppercase letter (`"J"`) is equivalent to `"shift-j"`. When a modifier is present (`"ctrl-J"`), the letter is taken literally — no implicit shift.

Action:

- Existing command name in kebab-case: `scroll-down`, `page-up`, `goto-line`, `toggle-line-numbers`, `mark-set`, `search-forward`, `shell-escape`, `hscroll-left`, `hscroll-right`, `anim-pause`, `anim-step-forward`, `anim-step-back`, `anim-restart`, etc. Unknown command names error at startup. `clipboard-yank-line` (copy the current line to the clipboard, requires `--clipboard`) is a valid command but left unbound by default so it doesn't clobber `y` (scroll-up).
- `!`-prefixed string: an inline shell command, run via the same infrastructure as `!cmd`.

Forbidden keys (cannot be rebound; error at startup): `m`, `'`, `-`, `Ctrl-X`, and digits `0 - 9`. These participate in multi-key sequences (marks, option prefix, jump-previous chord, numeric prefix accumulator) and rebinding them would break those features.

User bindings win over the built-in defaults. Any key not in the config keeps its default binding.

Log formats

`tess` ships with three built-in formats and reads user-defined formats from `~/.config/tess/formats.toml`. User entries with the same name as a built-in win.

Built-in formats

Name	Fields
<code>apache-common</code>	<code>ip</code> , <code>user</code> , <code>time</code> , <code>method</code> , <code>url</code> , <code>protocol</code> , <code>status</code> , <code>size</code>
<code>apache-combined</code>	<code>apache-common</code> + <code>referer</code> , <code>agent</code>
<code>nginx-combined</code>	same as <code>apache-combined</code>

Run `tess --list-formats` for the live list.

Defining your own

`~/.config/tess/formats.toml`:

```
# A simple level/message format.
[format.simple]
regex = '^(?P<level>\w+) (?P<msg>.*)$'

# A custom application log: timestamp, level, request id, message.
[format.app]
regex = '^(?P<ts>\S+ \S+) (?P<level>\w+) \[(?P<reqid>[0-9a-f]+)\] (?P<msg>.*)$'
# Optional default display template. CLI --display overrides.
display = '[<ts>] <level> <msg>'

# Override a built-in (here, a simplified apache-common variant).
[format.apache-common]
```

```
regex = '^(?P<ip>\S+) - - \[(?P<time>[^\]]+)\] "(?P<request>[^\"]+)" (?P<status>\d+) (?P<size>\S+)$'
```

Each format is one regex with named capture groups (`(?P<name>...)`). Field names become filterable. The regex must be anchored / specific enough that it matches only valid lines — non-matching lines are treated as “not parsed” and behave like filter mismatches.

Nested capture groups

Capture groups can be nested. The outer group captures the whole substring, the inner groups capture sub-parts, and **every named group becomes its own filterable field**. This is the cleanest way to expose a composite value (a timestamp, a URL, a version string) as both the full string *and* its parts:

```
# Apache CLF timestamp like "06/May/2026:16:13:44 +0200" — exposed both as
# `time` (the full bracketed string) and as discrete `year` / `month` / `day` /
# `hour` / `minute` / `second` / `tz` fields.
[format.apache-with-time-parts]
regex = '''^(?P<ip>\S+) \S+ \S+ \[(?P<time>(P<day>\d{2})/(?P<month>[A-Za-z]{3})/
(?P<year>\d{4}):(?P<hour>\d{2}):(?P<minute>\d{2}):(?P<second>\d{2})\s+(?P<tz>[+
-]\d{4}))\] "(?P<method>\S+) (?P<url>\S+) (?P<protocol>[^\"]+)" (?P<status>\d+) (?P<size>\S+)$'''
```

Then any of the parts can drive a filter:

```
tess --format apache-with-time-parts --filter year=2026 --filter 'hour~^1[0-2]$\naccess.log\n\ntess --format apache-with-time-parts --filter month=May access.log
```

Caveats:

- **Group names share one namespace** — every named group across the whole regex must be unique, even when nested.
- **The outer group still captures the literal substring**, including any delimiters inside it. Don’t put characters outside the outer group that you want included in `time`.
- **No post-processing** — the captured value is exactly what the regex matched. If a format has `May` you can’t filter on `05` either filter on `May` or use a regex predicate (`--filter 'month~^(May|05)$'`).
- For optional sub-parts (e.g. fractional seconds that may or may not be present), wrap the optional segment in a non-capturing group with `?: (?:\.(?P<micro>\d+))?`. When the segment isn’t there, the `micro` field simply won’t appear in the field map and any `--filter micro=...` won’t match — exactly what you want.

Display templates

A format can specify a default `display` template that reformats each parsed line into a chosen subset and order of fields. `--display TEMPLATE` on the command line overrides the format’s default. Either way, the syntax is the same:

Syntax	Meaning
<code><fieldname></code>	Replaced with the field’s captured value. Empty string if the regex didn’t capture the field on this line.

<code>\<</code>	Literal <code><</code> .
<code>\\</code>	Literal <code>\</code> .
<code>\x</code> (any other)	Left as <code>\x</code> (so you don't have to escape backslashes inside regex-like literals).
anything else	Literal.

Examples:

```
tess --format apache-combined --display ' [<status>] <method> <url>' access.log
tess --format app --display ' [<ts>] <level> <msg>' --filter 'level>=WARN' app.log
tess --format apache-combined --display '<status>: <url>' --filter 'status>=500' -
o errors.log access.log
```

Behavior notes:

- **Search runs against the rendered template**, so the highlight you see is the substring you typed. If you removed `<msg>` from the template, `/Renderer.php` won't find anything — that's intentional. Drop the template if you want raw-line search.
- **Filtering still operates on the raw captures**, not the rendered output.
`--filter status>=500` works regardless of whether `<status>` is in the template.
- **Lines that don't parse** against the format regex fall back to the raw line (no data is silently dropped), both in the interactive view and in `--output`.
- **Wrap-row scrolling** measures the rendered line length, so `j` walks the rendered wrap rows, not the raw line's.
- **Mutually exclusive with `--prettify`** (which already reshapes the byte stream).

How filtering works

`tess` runs the format's regex once per line. The named captures form a field map. Each `--filter` predicate looks up its field in that map and applies its operator. **All** predicates must match (AND). Lines that don't match the format regex at all are treated as non-matching.

In **hide mode** (default): non-matching lines are skipped entirely. The line counts, scroll, and `goto_bottom` operate on the matched-line numbering.

In **dim mode** (`--dim`): non-matching lines are still rendered, but with `Attribute::Dim` so they're visually faded; surrounding context stays readable. Useful for inspecting matches in their surroundings.

Global and local config

`tess` reads config in two layers:

1. **Global** — `/etc/tess/formats.toml` and `/etc/tess/keys.toml`, intended for system administrators on shared hosts who want every user to inherit a baseline set of formats, groups, and keybindings. Override the search path with `$TESS_GLOBAL_CONFIG_DIR=/path/to/dir` (useful for tests, CI, or non-standard installs).
2. **Local** — `~/.config/tess/formats.toml` and `~/.config/tess/keys.toml`, the per-user config that's been supported since 0.x.

Merge semantics

For each top-level entry — `[format.NAME]`, `[group.NAME]`, or a single binding key under `[bindings]` — if the local file declares it, the global entry for that exact key is replaced. Other entries at the same layer survive.

There is no per-field merging: overriding `[format.apache]` redeclares the entire section. (This matches how local configs override built-in formats today.)

Error handling

- A **missing** global or local file is silently ignored.
- A **malformed** global file prints a warning on stderr and is treated as empty; the binary continues with built-ins + local config.
- A **malformed** local file fails startup with a non-zero exit, naming the offending file.

Visibility

`tess --list-formats` annotates each format with its source:

<code>apache-common</code>	<code>[built-in]</code>	<code>ip, user, time, ...</code>
<code>apache-combined</code>	<code>[built-in]</code>	<code>ip, user, time, ...</code>
<code>internal-app</code>	<code>[global]</code>	<code>host, env, status</code>
<code>internal-app-override</code>	<code>[local, overrides global]</code>	<code>host, env, status</code>
<code>my-personal-format</code>	<code>[local]</code>	<code>user, action</code>

The override label names the **immediately replaced** layer. If a local entry shadows a built-in directly (no global in between), the label is `[local, overrides built-in]`.

Brew installs (macOS)

`/etc/tess/` works on macOS but requires `sudo` to populate. The brew formula does not drop config files into `/etc/tess/` operators who want a shared config on a macOS host either write there with `sudo` or set `$TESS_GLOBAL_CONFIG_DIR` to a brew-writable path (e.g. `$HOMEBREW_PREFIX/etc/tess/`).

Marks

Save and restore positions in the file. Marks are session-local: they live only as long as the running tess process.

Keys	Action
<code>m<x></code>	Set mark <code><x></code> to the current top line.
<code>'<x></code>	Jump to mark <code><x></code> . Silent no-op if the mark is unset.
<code>Ctrl-X Ctrl-X</code>	Jump to the previous position. Swaps current with previous, so pressing it twice returns to where you were.

`<x>` is any lowercase letter `a - z` or digit `0 - 9` (36 slots total).

The previous-position slot is updated automatically on every big jump: `search (/ , ? , n , N)`, `goto (Ng , NG , N%)`, `bare g`, `bare G`), mark jumps, and `Ctrl-X Ctrl-X` itself. Scrolling (`j`, `k`, `Space`, arrows) does NOT update it — so `Ctrl-X Ctrl-X` after scrolling around takes you back to wherever you last jumped from.

If a mark refers to a line that no longer exists (for example, after the source file has shrunk in `--live` mode), the jump lands at the last available line.

Multi-file navigation

Pass multiple files on the command line:

```
tess foo.log bar.log baz.log
```

The first file opens immediately. Use `:`-prefixed commands to navigate:

Command	Action
<code>:n</code> (or <code>:next</code>)	Next file. Shows <code>[no next file]</code> at the end.
<code>:p</code> (or <code>:prev</code>)	Previous file. Shows <code>[no previous file]</code> at the start.
<code>:b</code> (or <code>:buffers</code>)	Open the file picker overlay. Lists every file in the working set with its saved cursor position; type to filter (substring, case-insensitive); arrows/ <code>j</code> / <code>k</code> or scrollwheel (with <code>--mouse</code>) to move; Enter to switch; Ctrl-D to drop a file; Esc to close.
<code>:e <path></code> (or <code>:edit</code>)	Open <code><path></code> , append it to the file list, switch to it. <code>~/</code> expands to <code>\$HOME</code> .
<code>:f</code>	Show the current filename and position briefly in the status line.
<code>:q</code> (or <code>:quit</code>)	Quit (alias for <code>q</code>).
<code>:d</code> (or <code>:delete</code>)	Remove the current file from the list and switch to the next (or previous if at the end). Errors if only one file remains.
<code>:x</code> (or <code>:first</code>)	Jump to the first file in the list.
<code>:t</code> (or <code>:last</code>)	Jump to the last file in the list.
<code>:help</code> (or <code>:h</code>)	Open the help overlay listing every keybinding, grouped by category. <code>F1</code> also opens it. Type to filter; arrows/scrollwheel (with <code>--mouse</code>) to scroll; Esc to close (clears filter first if non-empty).
<code>:tag NAME</code>	Jump to the named ctags/etags tag. See Tag jumping.
<code>:tnext</code> / <code>:tprev</code>	Cycle through multiple matches for the current tag.
<code>:incsearch</code>	Toggle incremental search on/off. Takes effect on the next search prompt. See Search.
<code>:yank</code>	Copy the current line to the system clipboard (requires <code>--clipboard</code>). See Clipboard.

Press `:` to enter the colon prompt; type the command and press Enter, or press Esc to cancel. Bare Enter on an empty prompt dismisses without running anything.

When the file list has more than one entry, the status line shows `<label> [N/M]` next to the filename. Custom `--prompt` templates can place this anywhere via the `<file-index-tag>` placeholder.

State preserved across file switches

- Marks (`m a`, `'a`) are session-wide. Setting `m a` in `foo.log` and switching to `bar.log` doesn't lose the mark — `'a` returns to that position in `foo.log`.
- The previous-position slot (`Ctrl-X Ctrl-X`) tracks across files too. Jump from line 50 in `foo.log` to line 1 in `bar.log` (via `:n`), press `Ctrl-X Ctrl-X`, return to `foo.log:50`.
- The active search regex persists. After `:n`, pressing `n` searches the new file from the top with the same pattern.

State reset on every switch

- Top-of-screen position resets to line 1. Each file starts fresh.
- The numeric-prefix accumulator, mark-pending, and `Ctrl-X`-pending states all clear.

Stdin and multi-file

Piped stdin (`cat foo | tess bar.log baz.log`) still wins: stdin is read as the only source, and file arguments are ignored with a warning. Multi-file navigation requires file-path arguments.

`--follow` and `--live` with multi-file

If you invoked with `--follow`, each file you switch to enters follow mode. If you invoked with `--live`, each file is opened as a live source.

Tag jumping

`tess` can read a `ctags` - or `etags` -format tags file and jump to named definitions, vim-style.

Loading a tags file

Flag	Behavior
<code>-T PATH / --tag-file PATH</code>	Use this exact tags file.
(default)	Walk up from the first file's directory looking for <code>tags</code> .

If `-t` is given but no tags file is found, `tess` exits with code 1 and a message on stderr.

Jumping

Action	Trigger
Jump to a tag at startup	<code>tess -t NAME [files...]</code>
Open the tag-name prompt at runtime	<code>Ctrl-]</code> (then type a name, Enter)
Jump to a tag via colon prompt	<code>:tag NAME</code>
Pop the tag stack	<code>Ctrl-T</code>
Next match (multi-match tag)	<code>:tnext</code>
Previous match	<code>:tprev</code>

The tag stack is a vim-style LIFO of jump-from positions. Each `:tag / Ctrl-]` push the current (file, line) onto the stack; `Ctrl-T` pops back. The stack is unbounded.

Multi-match cycling

When a tag has more than one definition (e.g. an overloaded function in multiple files), the first match is shown and a status indicator appears at the right end of the status line:

```
[tag: foo (1/3)]
```

`:tnext` advances the cursor; `:tprev` retreats. The indicator stays visible until you `Ctrl-T` pop the stack or start a new tag jump. Other movement (`j` / `k` / `/` etc.) does not clear it.

Single-match jumps do not show the indicator.

Supported tag-file formats

- **ctags traditional:** `name\tnfile\taddress`
- **ctags exuberant / universal:** same plus optional `;"` extended fields (kind, scope, etc.). Extended fields are ignored.
- **etags** (Emacs): standard etags format with `\x0c\n`-separated sections.

Tag addresses can be line numbers (`42`) or vi-style search patterns (`/^fn foo()$/` or `?pattern?`). Pattern addresses are converted to regex anchors (`^...$` preserved); other metacharacters in the pattern body are escaped.

If a pattern address doesn't match in the source file, `tess` shows `[tag pattern not found]` and stays where it is — the stack push already happened, so `Ctrl-T` returns to where you came from.

Custom prompt placement

The `<tag-tag>` placeholder in `--prompt` templates resolves to `[tag: NAME (N/M)]` when a multi-match cycle is active, empty otherwise.

Completion

In the `:tag / Ctrl-]` prompt, `Tab` autocompletes the buffer to the longest common prefix of matching tag names. A second consecutive `Tab` shows the match count in the prompt hint. Typing or `Backspace` clears the cached match list so the next `Tab` re-queries.

Auto-reload

The tags file is re-stat'd before every tag operation (`:tag NAME`, `Ctrl-]`, `:tnext` / `:tprev` / `:tselect`, `Tab` completion). When the mtime has advanced, the file is re-parsed in place and a transient `[tags reloaded]` status hint is shown. Run `ctags` in another terminal and the next `Ctrl-]` will see the fresh entries.

`:tselect` — picker for multi-match tags

`:tselect NAME` looks up the tag and opens a picker overlay listing every match (`N. <file>:<addr>`). Use `↑` / `↓` or `j` / `k` to navigate, `Enter` to jump, `Esc` to cancel. Number keys `1 – 9` pick directly when there are fewer than 10 matches.

`:tselect` with no arg uses the currently-active multi-match list (the one `:tnext` / `:tprev` cycle through), so you can switch from cycling to direct-select mid-session.

Chained ; addresses

`tess` recognizes ctags addresses chained with `;;` `/^outer_anchor$/;inner_pattern/`. The second step searches starting from the line matched by the first, matching vim's behavior.

`;` inside `/.../` or `?...?` is treated as literal so patterns containing semicolons are preserved.

Unsupported address forms (`:s/foo/bar/`, `:call ...`, etc.) jump to line 1 of the target file and surface a `[tag address not supported: \<raw\>]` status hint instead of silently failing.

Color and ANSI escapes

`tess` interprets ANSI SGR escape sequences (colors, bold, italic, underline, dim, reverse, strike) and OSC 8 hyperlinks by default. Non-SGR CSI sequences (cursor moves, screen clears, mouse mode setup) are parsed and discarded silently so they can't corrupt the pager layout.

This means colored output flows through cleanly:

```
ls --color=always | tess
git --no-pager diff --color=always | tess
bat --color=always file.rs | tess
```

The 16 named colors, xterm-256 indexed colors, and 24-bit truecolor are all recognized.

Opting out

Flag	Behavior
<code>--no-color</code>	Show raw control bytes as <code>^X</code> glyphs (pre-0.18 behavior).
<code>-r</code> / <code>--raw-control-chars</code>	Pass every byte verbatim to the terminal — including cursor moves. Risky: long-line wrap math may break. <code>less-style -r</code> .

The `NO_COLOR` environment variable (any non-empty value) and `CLICOLOR=0` also force `--no-color` behavior. Explicit flags always win over env.

Switching modes at runtime

Press `:` then type `color` to cycle through the three ANSI policies: `strict` → `interpret` → `raw`. Or pass an explicit mode (`:color strict`, `:color interpret`, `:color raw`). Useful when a colored log is unreadable in `raw` and you want to flip back to `interpret` without restarting.

Truecolor (24-bit RGB)

`--truecolor=auto` (default) checks the `COLORTERM` environment variable and downsamples 24-bit RGB colors to the xterm 256-color palette when truecolor isn't advertised.

`--truecolor=never` always downsamples; `--truecolor=always` passes RGB through regardless of terminal advertising.

Theming the status line and prompt

`--status-style` styles the status row at the bottom (default `reverse` — matches pre-0.23 behavior). `--prompt-style` styles the row when a custom `--prompt` template is active. Per-format `prompt_style = '...'` in `formats.toml` provides a per-format default; CLI

`--prompt-style` wins.

Grammar: comma-separated tokens.

Token	Meaning
<code>bold</code> , <code>dim</code> , <code>italic</code> , <code>underline</code> , <code>reverse</code>	SGR attributes.
<code>fg=COLOR</code> , <code>bg=COLOR</code>	Foreground / background.
<code>COLOR</code> = named (<code>black</code> .. <code>white</code> , optional <code>bright-</code> prefix), <code>#RRGGBB</code> , or 0–255.	

Examples:

```
tess --status-style 'bold,fg=cyan,bg=black' app.log
tess --prompt '<label> <pct>%' --prompt-style 'fg=bright-blue' app.log
```

Embedding SGR in display / prompt templates

`--display` and `--prompt` literals accept backslash-escape control bytes: `\e` / `\x1b` / `\033` (ESC), `\n`, `\t`, `\r`, plus `\xHH` and `\NNN` (hex / octal byte). The bytes flow into the normal render pipeline, so they're interpreted when ANSI mode is `interpret` or passed through when `raw`:

```
tail -f app.log | tess --format apache-common \
  --display '\e[33m<time>\e[0m \e[1m<status>\e[0m <msg>'
```

What patterns see

`/search`, `--filter`, and `--grep` always match against the SGR-stripped visible text — even in default `Interpret` mode. So `/error` finds 'error' in a red `\x1b[31merror\x1b[0m` regardless of whether colors are showing.

Color state across lines

SGR state persists across newlines until explicitly reset (just like a terminal). When scrolling backward, `tess` walks up to 256 lines back looking for a reset point and replays state forward; if no reset is found within the cap, the first visible lines may show default colors until a reset appears.

Output to non-TTY

When stdout is a file or pipe (`tess --stdout`, `-o file`, `| grep`), ANSI escapes are stripped automatically. Use `--no-color` to also strip when sending TTY output that you'll forward to a non-color-aware tool.

Running shell commands

Press `!` inside the pager to enter a shell command. The prompt shows the command as you type it; `Enter` runs it, `Esc` cancels.

When the command runs, `tess` drops the alt-screen and raw mode so the command sees a normal interactive terminal. Output streams directly to your terminal. After the command exits, `tess` prompts:

```
[Press any key to continue]
```

After you press any key, `tess` restores the alt-screen and redraws. Interactive commands like `!vim notes.txt` work normally.

The shell is taken from `$SHELL`, falling back to `/bin/sh`.

Hex display

Render the source as an `xxd`-style hex dump. One row covers 16 bytes: an 8-digit hex offset, the bytes themselves grouped in 8×2 -byte words, and an ASCII gutter where printable bytes appear and everything else shows as `. .`.

```
tess --hex /usr/bin/ls
tess -f --hex /var/log/binary-feed.bin # follow mode works
```

`--hex` is mutually exclusive with `--filter`, `--grep`, `--prettify`, `--format`, `--display`, `--record-start`, and `--prompt` — hex mode is fundamentally byte-level and these features are line- or record-oriented.

Search (`/pattern`) inside hex mode operates on the rendered row text, so you can find ASCII strings in the gutter or hex byte sequences:

```
/Hello      # find the ASCII string in the gutter
/4865 6c6c  # find the same bytes by their hex
```

Images

`tess` detects image files by magic bytes (PNG, JPEG, GIF, BMP, WebP, TIFF, TGA, ICO, PNM) and renders them as colored ASCII art. No flags required — just open the file:

```
tess photo.png
tess logo.gif # animated GIFs play automatically
```

Flags

- `--no-image` — skip rendering and show the raw bytes instead (combine with `--hex` for a byte-level view).
- `--blocks` — Unicode half-block (`▀`) mode. Each terminal cell encodes two pixel rows (top as fg, bottom as bg), doubling vertical resolution. Ignored when a non-ASCII `--image-protocol` is active.
- `--image-width N` — render at N columns (protocol modes treat it as a fit-width override). Defaults to the current terminal width.
- `--no-animate` — open an animated image as its static first frame instead of playing it. See **Animation** below.
- `--image-protocol auto|kitty|sixel|ascii` — choose how images are drawn. Default `auto`:
 - `auto` queries the terminal for graphics support (a Kitty graphics query, plus a DA1 query for Sixel) and falls back to the colored-ASCII / half-block path when neither is available. Explicit `kitty` / `sixel` / `ascii` skip detection. When both protocols are detected, Kitty is preferred over Sixel.
 - In `kitty` / `sixel` mode the image fits the terminal width and you scroll down through a tall image with the usual vertical-scroll keys; `←` / `→` horizontal scroll is a no-op in protocol mode (it still works in `ascii` mode wider than the terminal). The status line shows `[kitty]` / `[sixel]` while in image mode.
 - Terminal support: the Kitty graphics protocol works on Kitty, iTerm2, WezTerm, and Ghostty; Sixel works on foot, xterm (built with `--enable-sixel`), mlterm, and WezTerm.

- Combined with `-o FILE / --stdout`, an explicit `kitty` or `sixel` writes the raw encoded escape-sequence bytes to the file or pipe (e.g. `tess --image-protocol sixel --stdout photo.png > out.sixel`); `auto` / `ascii` export colored ASCII art as described below.

Animation

Animated images **play automatically** when you open them. Animated GIF is the primary, fully-tested case; APNG and animated WebP are also decoded and played where the underlying `image` crate supports them.

Playback works in **every render mode**. The colored-ASCII / `--blocks` half-block path animates on any terminal (Warp included); the `--image-protocol kitty / sixel` paths animate on graphics-capable terminals by re-emitting each frame. (There is no native Kitty animation protocol — every mode uses per-frame re-emit.)

The animation **honors the GIF loop count**: after the requested number of loops it rests on the last frame. A loop count of `0` or absent — the common case — loops forever. WebP / APNG loop counts are not parsed and are treated as infinite.

Transport keys control playback. They are active globally but do nothing when the current view has no animation, and they are all remappable in `~/.config/tess/keys.toml`:

Key	Action	Binding name
<code>p</code>	Pause / resume (also revives a finished animation)	<code>anim-pause</code>
<code>.</code>	Step one frame forward (auto-pauses)	<code>anim-step-forward</code>
<code>,</code>	Step one frame back (auto-pauses)	<code>anim-step-back</code>
Backspace	Restart from the first frame	<code>anim-restart</code>

The status line in image mode shows a transport badge: `[play i/n]` while playing, `[pause i/n]` when paused, and `[done n/n]` once a finite loop count has finished.

Pass `--no-animate` to open the static first frame instead of playing.

If a source is genuinely animated but its frames can't be decoded — most notably a **16-bit APNG**, which the underlying decoder rejects — `tess` shows the static first frame and flashes `couldn't decode animation; showing first frame` on the status line, rather than silently dropping the animation. Re-encode such an APNG as 8-bit to play it.

Batch export is always a static image. Combining an animated source with `-o FILE / --stdout` emits a single frame — animation is interactive-only.

Color and export

Color output uses 24-bit truecolor SGR. Pass `--no-color` for a plain-text render using the character ramp (`@#%*+=- .`) or block-shade characters (`▀▁▂▃▄▅▆▇█▉▊▋▌▍▎▏▐░▒▓▔▕▖▗▘▙▚▛▜▝▞▟■□▢▣▤▥▦▧▨▩▪▫▬▭▮▯▰▱▲△▴▵▶▷▸▹►▻▼▽▾▿▿▰▱▲△▴▵▶▷▸▹►▻▼▽▾▿`) with no SGR codes.

Export the rendered art with `-o FILE` or `--stdout`:

```
tess --stdout photo.png > art.txt           # plain text (redirected stdout → no
ANSI)
tess --stdout photo.png | cat               # same
tess --stdout --no-color photo.png         # force plain text
```

Cargo feature

The `image` feature is enabled by default. Build without it to produce a smaller binary that treats all inputs as text:

```
cargo build --release --no-default-features
```

Preprocessing input

Pipe the source file through an external command before `tess` reads it. Useful for viewing PDFs as text (`pdftotext`), tar archives as listings (`tar -tzvf`), and so on.

Two ways to set the preprocessor:

```
# Per-invocation (CLI flag):
tess --preprocess '|pdftotext %s -' document.pdf

# Persistent (env var, less-compatible):
export LESSOPEN='|lesspipe.sh %s'
tess any-file.gz
```

CLI flag overrides env var. `--no-preprocess` ignores both for one invocation.

Syntax: the command must start with `|` (pipe mode). `%s` is substituted with the file path (shell-quoted to handle spaces). The command's stdout becomes `tess`'s input.

On failure (non-zero exit, missing executable, empty output), `tess` falls back to the raw file and shows `[preprocess-failed: <stderr>]` in the status line. Custom `--prompt` templates can include the failure tag via `<preprocess-failed-tag>`.

Mutually exclusive with `--hex`, `--follow`, and `--live`.

Stdin pipes (`cat foo.txt | tess`) skip preprocessing entirely.

Multi-line records

Some log formats emit records that span many physical newlines: PHP error logs with stack traces, Java exception traces, multi-line debug payloads. Set `record_start` to a regex that matches the first line of each record, and `tess` treats every following non-matching line as a continuation:

```
[format.php-app]
record_start = '^\[ '
regex       = '^\[ (?P<ts>[^\]]+)\] (?P<level>\w+) (?P<msg>[\s\S]+)$'
display     = '[<ts>] <level> <msg>'
```

Then run:

```
tess --format php-app /var/log/php/error.log
```

Or use the CLI flag directly without a format:

```
tess --record-start '^\[ ' /var/log/php/error.log
```

When records mode is active:

- Search (/pattern , n , N) matches against the full record bytes with embedded newlines. Use (?s) if you want . to match \n .
- --filter FIELD<op>VALUE matches against the parsed record (so [\s\S]+ capture groups span lines).
- --grep PATTERN matches against the full record bytes.
- Hide mode hides every line of a non-matching record; dim mode dims them but keeps them visible.
- Status line shows L<line>--<line>/<total> R<rec>--<rec>/<total> .
- Ng jumps to physical line N; NG jumps to record N; N% jumps to N percent through the file by bytes. Esc cancels a partially-typed numeric prefix.

Lines before the first record_start match are collected into a single synthetic record numbered 0. If no line in the file matches the regex, the entire content becomes one big record (this is usually a sign that the regex is wrong).

--head N and --tail N continue to count physical lines, not records.

Examples

Plain file viewing

```
# Open a file
tess README.md

# Show line numbers and disable line wrapping
tess -N -S src/main.rs

# Custom tab width for code with mixed indentation
tess --tab-width 4 Makefile
```

Piped input

```
# Page through git log
git log | tess

# Page a colored command's output (tess passes ANSI through faithfully –
# control bytes render as ^X, so use a tool that strips them if you want
# them gone)
ls --color=always | tess

# Build output, kept on screen for inspection
cargo build 2>&1 | tess
```

Quick first/last N

```
# Last 100 lines of a 5 GB log – opens instantly
tess --tail 100 /var/log/access.log

# First 50 lines of a generated file
tess --head 50 schema.sql
```

```
# Last 1000 lines and follow new ones (the headline log-watching command)
tess -f --tail 1000 /var/log/access.log
```

Following live output

```
# Watch a log file
tess -f /var/log/syslog

# Watch a finite producer (returns to shell when the producer ends)
( for i in $(seq 1 200); do date; sleep 0.5; done ) | tess -f

# In tess, press Shift-F to pause auto-scroll, scroll back to read context,
# press G to jump back to the live tail, Shift-F again to re-engage.
```

Apache log analysis

Sample log line (apache-combined):

```
127.0.0.1 - alice [10/Oct/2023:13:55:36 +0000] "GET /index.html HTTP/1.1" 200 2326
 "-" "Mozilla/5.0"
```

```
# Only 5xx errors
tess --format apache-combined --filter status~^5 access.log

# Only 5xx errors on /api/* paths (multi-filter AND)
tess --format apache-combined --filter status~^5 --filter url~^/api/ access.log

# Everything except 200s (single-quote because of bash's history expansion on `!`)
tess --format apache-combined --filter 'status!=200' access.log

# Errors from a specific subnet
tess --format apache-combined \
  --filter status~^[45] \
  --filter ip~10\.0\. \
  access.log

# Exclude bot traffic (single-quoted to escape the `!`)
tess --format apache-combined --filter 'agent!~bot' access.log

# Show errors with surrounding context (dim non-matches instead of hiding)
tess --format apache-combined --filter status~^5 --dim access.log

# Tail-follow only the errors as they happen
tess -f --tail 100 --format apache-combined --filter status~^5 access.log
```

Custom format from your own app

```
# 1) Define it in ~/.config/tess/formats.toml:
mkdir -p ~/.config/tess
cat > ~/.config/tess/formats.toml <<'EOF'
[format.app]
regex = '^(?P<ts>\S+ \S+) (?P<level>\w+) \[(?P<reqid>[0-9a-f]+)\] (?P<msg>.*)$'
EOF
```

```
# 2) Verify it shows up
tess --list-formats

# 3) Use it
tess --format app --filter level=ERROR app.log

# 4) ERRORs and WARNs (regex-OR via alternation)
tess --format app --filter level~^(ERROR|WARN)$ app.log

# 5) Errors for a specific request id
tess --format app --filter level=ERROR --filter reqid=deadbeefcafe app.log

# 6) Watch ERRORs land in real time
tess -f --tail 200 --format app --filter level=ERROR app.log
```

Filtering piped output

```
# Synchronous stdin (no -f): tess buffers everything, then filters
journalctl --no-pager | tess --format apache-combined --filter status~^5
# (won't actually match – journald isn't apache, but the shape is the same:
# pipe in, parse, filter)

# Streaming stdin with -f: tess reads as a background thread
tail -F /var/log/access.log | tess -f --format apache-combined --filter status~^5

# Note: --tail isn't supported with streaming stdin (no random access).
# Use the file form when you want both --tail and -f together.
```

Combining everything

```
# Tail-follow the last 5000 lines, only 5xx errors, with line numbers,
# don't wrap long URLs:
tess -f --tail 5000 -N -S \
  --format apache-combined \
  --filter status~^5 \
  /var/log/access.log

# Same with context preserved (dim mode):
tess -f --tail 5000 -N -S \
  --format apache-combined \
  --filter status~^5 \
  --dim \
  /var/log/access.log
```

tess --list-formats

```
$ tess --list-formats
apache-combined: ip, user, time, method, url, protocol, status, size, referer,
agent
apache-common: ip, user, time, method, url, protocol, status, size
nginx-combined: ip, user, time, method, url, protocol, status, size, referer,
agent
```

(User-defined formats appear in the same list.)

Groups: command-line shortcuts

Repeating long invocations gets tiresome. A `[group.NAME]` entry in

`~/.config/tess/formats.toml` defines a shortcut: when you pass `--NAME` on the command line, `tess` expands it into a fixed set of flags (format, file, follow, tail, head, dim, line numbers, chop, tab width, default filters). Bare positionals after the group token become **filters**.

Example

```
# ~/.config/tess/formats.toml

[format.errorlog]
regex = '^(?P<ts>\S+ \S+) (?P<level>\w+) \[(?P<reqid>[0-9a-f]+)\] (?P<msg>.*)$'

[group.errorlog]
format = "errorlog"
file = "/var/log/apache2/SE.error"
follow = true
tail = 1000
filter = ["level=ERROR"] # optional: pre-applied filters

[group.access5xx]
format = "apache-combined"
file = "/var/log/apache2/access.log"
follow = true
filter = ["status~^5"]
```

With this config:

```
# Watch ERRORS in the app log:
tess --errorlog
# Equivalent to:
tess --format errorlog --follow --tail 1000 --filter 'level=ERROR' /var/log/
apache2/SE.error

# Add an extra filter on the fly – positionals become --filter args:
tess --errorlog 'msg~timeout'
# Equivalent to the above plus --filter 'msg~timeout' (ANDed).

# Multiple ad-hoc filters:
tess --errorlog 'msg~timeout' 'reqid=deadbeefcafe'

# Override a group flag with a CLI flag (the CLI value wins):
tess --errorlog --tail 50 'msg~timeout'
# Group has tail=1000 but you override to 50.
```

Group fields

All optional. Anything left out simply isn't passed.

Key	Type	Maps to CLI flag
<code>format</code>	string	<code>--format <name></code>

file	string	positional FILE
follow	bool	-f / --follow
tail	integer	--tail N
head	integer	--head N
dim	bool	--dim
line_numbers	bool	-N
chop	bool	-S
tab_width	integer	--tab-width N
display	string	--display <template>
filter	array of strings	--filter X (one entry per element)
grep	array of strings	--grep X (one entry per element)
or_filter	array of strings	--or-filter X — default OR-group (one entry per element)
or_grep	array of strings	--or-grep X — default OR-group (one entry per element)

A group's `display` template is rendered just like the `--display` flag, so it needs the group (or a CLI flag) to also set a `format`. A `--display` typed on the command line after the group token overrides the group's template:

```
[group.errs]
format = "myapp"
filter = ["level=ERROR"]
display = "<time> <level> <msg>" # rendered unless a CLI --display overrides
```

Named OR-groups in config

Top-level `or_filter` / `or_grep` arrays belong to the implicit **default** OR-group. Named OR-groups are defined as `[group.NAME.or.<subname>]` sub-tables, each with its own `filter` and/or `grep` arrays. Every non-empty OR-group (default or named) must have at least one match for the record to be visible. Named groups give you independent OR-buckets that are AND'd together.

```
[group.intrusion]
format = "apache-combined"
filter = ["status~^4"] # required: only 4xx responses
or_grep = [' /\.env', '/wp-login', '/admin'] # default OR-group: suspicious paths

[group.intrusion.or.methods]
filter = ["method=POST", "method=PUT"] # named OR-group "methods"
```

`apache-combined` is a built-in format, so every `status` / `method` field reference is valid. With this group, `tess --intrusion access.log` shows lines where:

- `status~^4` (required AND — a 4xx response),
- at least one suspicious path matches (default OR-group), AND
- the method is `POST` or `PUT` (`methods` OR-group).

Override semantics

When the group is expanded, its flags appear in `argv` before any flags you typed *after* the group token. For repeatable flags (`--filter` , `--grep`), CLI values **add** to the group's. For single-value flags (`--tail` , `--head` , `--tab-width` , `--format`), the **last occurrence wins**, so a CLI flag after the group token overrides the group's value.

Groups also support a `grep` field that mirrors `filter` . Each entry becomes a repeated `--grep <pattern>` after group expansion, and the user's own `--grep` arguments accumulate on top:

```
[group.errorlog]
format = "errorlog"
follow = true
filter = ["level=ERROR"]
grep = ["timeout", "deadlock"] # both patterns must match
```

Restrictions

- A group cannot be named the same as a built-in flag (`format` , `filter` , `dim` , `head` , `tail` , `follow` , `LINE-NUMBERS` , `chop-long-lines` , `tab-width` , `list-formats` , `help` , `version`). Trying to load such a group prints an error and exits.
- Once a group token is seen, every subsequent bare positional in `argv` (anything that doesn't start with `-`) becomes a `--filter` argument. To open a different file alongside an active group, edit the group or define a second one — there is no `--file` override flag yet.

Layouts: saved split arrangements

A `[group.NAME]` shortcut configures a *single* view. A `[layout.NAME]` table goes one step further and saves a whole **split**: an orientation plus an ordered list of panes, each pane configured exactly like a group. Open it with `--<layoutname>` on the command line, or `:layout NAME` at runtime.

```
[layout.ops]
orientation = "vertical" # "vertical" (default, side-by-side) or
                        "horizontal" (stacked)

[[layout.ops.pane]]
file = "/var/log/app.log"
format = "myapp"
filter = ["level=ERROR"]
follow = true

[[layout.ops.pane]]
file = "/var/log/nginx/access.log"
format = "nginx-combined"
grep = ["\\b5\\d\\d\\b"]
```

`tess --ops` opens both panes side by side: the app error log on the left (following), the nginx 5xx lines on the right. `:layout ops` does the same to the current session, replacing whatever was on screen.

A layout compiles down to the `--` per-pane form plus the orientation — so it is exactly equivalent to typing each pane's flags out by hand and passing `--split` (or `--hsplit` for `orientation = "horizontal"`).

Pane fields

Each `[[layout.NAME.pane]]` reuses the **same field vocabulary as a group** (see Group fields and Named OR-groups in config) — `format`, `filter`, `grep`, `or_filter`, `or_grep`, `display`, `follow`, `tail`, `head`, `dim`, `line_numbers`, `chop`, `tab_width`, and `[layout.NAME.pane.or.<subname>]` sub-tables — **plus a required file**. A pane with no `file` is an error at load time.

Rules and restrictions

- **Orientation is flat.** `vertical` (the default) lays panes side by side; `horizontal` stacks them. There is no nested grid — every pane shares one orientation.
- **Names must be disjoint from groups** and must not shadow a built-in flag, same as `[group]` names. A layout that collides with a group name errors at startup.
- **--<layoutname> is mutually exclusive** with `--split`, `--hsplit`, `--diff`, `--gitdiff`, and the `--right-*` pane flags. Pick a layout *or* an ad-hoc split, not both.
- **Give --<layoutname> alone.** Like a group token, once the layout token is seen the remaining argv is consumed building its panes. A conflicting flag placed *after* `--<layoutname>` is folded into the last pane rather than rejected; put any standalone flags *before* the layout token, or better, keep the invocation to just `tess --<layoutname>`.

Files

- `~/.config/tess/formats.toml` — user-defined log formats, groups, and layouts. See Defining your own, Groups, and Layouts.

Exit codes

Code	Meaning
0	Clean exit
1	Startup error (bad arguments, file not found, not a regular file, no input on a TTY)
2	Runtime error (e.g. invalid filter spec, terminal init failure)

Common pitfalls

- **bash: !=200: event not found** (or `!~notice` etc.) — the `!` in negating filter operators triggers shell history expansion. Single-quote the filter: `--filter 'status!=200'`. See the note under `--filter` in Command-line flags.
- **tess --mygroup somefile.log doesn't open somefile.log** — when a group is active, bare positionals are treated as filters, so `somefile.log` becomes `--filter somefile.log` and fails to parse (no operator). The group's `file = "..."` is the file. To view a different file, drop the group flag.
- **--filter without --format** — errors out with `tess: --filter requires --format`. Pick a format first; use `--list-formats` if unsure.

- **Filter field doesn't exist in the format** — errors out before entering the pager with the available field list, e.g.
`field 'foo' is not in format 'apache-combined' (available: ip, user, time, method, url, protocol, status, size, referer, agent)`.
 - **Lines that don't parse against the chosen format** — treated as non-matches. Hidden by default; visible-but-dimmed with `--dim`. If many lines aren't parsing, your regex is probably too strict.
 - **--tail on streaming stdin (-f with no file)** — prints
`tess: --tail is not supported on streaming stdin (-f); ignoring` and continues without it.
 - **Pipeline doesn't return to shell after q** — you ran something like
`(while true; do ...; done) | tess -f`. The producer subshell is in an infinite loop; bash waits for it. Use a finite producer or open the file directly.
 - **Big regex on huge files in hide mode is slow at startup** — hide-mode filtering does one full index pass before the first frame to find matches. For non-filtered or dim-mode viewing, indexing is lazy and 10 GB files open instantly.
-

Glossary

- **Logical line** — one newline-bounded record. The line numbering used by `--head`, `--tail`, `goto`, `scroll`, etc.
 - **Display row** — one row on the terminal. A long logical line wraps into several display rows when wrap is on.
 - **Source** — a `tess` byte source: a file (mmap-backed with a streaming companion handle), synchronous stdin, or streaming stdin.
 - **Hide mode / dim mode** — what `--filter` / `--grep` does to non-matching lines. Hide is the default.
-

Versions

This manual targets `tess 0.21.0`. Run `tess --version` to confirm.