QuickNotes

User Manual

A lightweight, cross-platform text editor that pays homage to the classic DOS word processor XyWrite.

The QuickNotes Project

Built 2026-06-06T06:08:09Z

Contents

Preface

quick (kwik), *adj.*
1. immediate, instantaneous, rapid
2. lively, agile, snappy

"Quick, quick, good hands."
— Shakespeare, *Antony and Cleopatra*

QuickNotes is a single-file Python / Tkinter Markdown note-taking application with an embedded scripting language, a user-extensible library system, a press-a-key facility for binding commands or scripts to a key, and many other useful features. In addition to the familiar menu interface, it offers the unique feature of a command line (the *CMline*), which allows you to drive the program using short, mnemonic commands.

The result is a note-taker that grows with you. On Day One you use it as a Markdown or text editor with a few keyboard shortcuts. A few weeks in, you get comfortable with the CMline and start chaining commands together. A few months in, and your `user.qnu` library file puts a personal toolkit of essential routines at your fingertips, accessible via the CMline or by the keyboard assignment of your choice.

QuickNotes was inspired by the classic DOS word-processor XyWrite, and adopts a good deal of XyWrite's pithy command syntax. Likewise, QuickNotes' Programming Language – QPL – owes a debt to XPL – the XyWrite Programming Language – but modernizes it by introducing plain-text scripts, intuitive keywords, and easy-to-learn command patterns. QPL also gives you access to the power of the underlying operating system: you can execute

shell commands and direct their output to a QPL variable, which can then be used by your script.

This manual is intended for both new users and those experienced in the use of text editors. Chapters one through six are a friendly user's guide: they walk a new user from "what is this program" through writing notes, searching, replacing, and driving the program from its command line. The remaining chapters read more like a reference work, organized for lookup rather than reading cover-to-cover. The appendices collect indexes, lists of commands and special variables, and a small gallery of useful scripts.

The date stamped on this manual's cover page is the QuickNotes revision the manual was built against — the same `Revision:` string that appears at the top of `quicknotes.py` and in the *About* dialog. When the two disagree, trust `quicknotes.py`; an out-of-date manual may describe verbs or behavior that have since changed.

Chapter 1

Overview and Quick Start

This chapter introduces the program, gets you to your first saved note, and finishes with a single-line script — enough to give you a sense of where the depths lie before later chapters take you down into them.

1.1 Running QuickNotes

QuickNotes is a single Python file (`quicknotes.py`). To run it you'll need Python 3.9 or later with Tkinter included — the default in essentially every Python distribution except some headless Linux server builds, where Tkinter is a separate `python3-tk` package.

Locate `quicknotes.py` in a directory of your choice. To launch QuickNotes, open a terminal, navigate to the directory that contains `quicknotes.py`, and command:

```
python quicknotes.py
```

On Windows you can also double-click the file in Windows Explorer and Python's file association will launch it. On macOS you may need `python3` instead of `python`.

The first time you launch it, QuickNotes creates two folders next to

`quicknotes.py`: a `notes/` folder for your note files, and a `scripts/` folder for `.qns` scripts and `.qnu` libraries. It also creates a JSON config file, located in the directory with `quicknotes.py` . Deleting any of these is harmless — they're recreated on next launch.

1.2 First launch

You'll see a single window with an empty editor tab named `(new)`, a tab strip across the top, a menu bar (File, Edit, Search, View, Tools, Options, Scripts, Help), and — importantly — two stacked rows along the bottom of the window. The upper row is the **CMline**, where you type commands; the row below it is the **status line**, where QuickNotes reports the result. The CMline is where QuickNotes really comes alive.

Focus moves between the text window and the CMline with **F5** (jump to the CMline) and **F10** (toggle between text and CMline). The status line directly beneath the CMline is where one-line feedback (`found`, `saved`, `no match`, error messages) appears. Each row takes the full window width, so long file paths and multi-clause error messages stay readable.

A note on conventions

Throughout the manual, keys are written like F5 or Ctrl+S. Command-line commands are shown like `PR hello`. Anything you'd literally type into the CMline or a script appears in a fixed-width block.

1.3 A ten-minute tour

We're going to write a short note, save it, search inside it, run a one-line CMline command, and finish with a small script. By the end you'll have used every major piece of the program at least once.

1.3.1 A. Write something

Click in the editor pane and type a few lines:

```
Things to remember
==================

- Buy coffee.
- Email Alex about Tuesday's meeting.
- Read chapter 3 of the QuickNotes manual.
```

QuickNotes is a Markdown editor, so the == underline made the first line a heading and the − lines became a bullet list — but the file is still just plain text. *View → Toggle Markdown preview (Ctrl+P)* shows the rendered version side-by-side when you want it.

1.3.2 B. Save it

Press **Ctrl+S** (or *File → Save*). The first save of a new note prompts for a title; type **Things to remember** and press Enter. The tab name changes from `(new)` to your note's title.

QuickNotes saves notes as plain `.md` files in your `/notes` folder, with a small header at the top recording the title, the creation date, and any tags. You can open the file in any other Markdown editor; it's just text.

1.3.3 C. Search it

QuickNotes gives you two ways to search:

- Press **Ctrl+F**, type "coffee", and press **Enter** to search.
- Press **F5** to enter the CMline. Type:

```
SE /coffee/
```

and press Enter. The cursor jumps to the first occurrence of "coffee" in

your note and the status line reports `found`. `SE` (Search) is the simplest of QuickNotes' search verbs — it takes a literal pattern between two separator characters. You used slashes; anything that isn't already in your pattern works (`SE !coffee!`, `SE ,coffee,`, …).

You can chain commands on one line with semicolons. Still in the CMline, try:

```
SE /Alex/; PR Found a name.
```

`PR` (think "Print" or "Prompt") writes a message to the status line. This is the first hint that the CMline is more than a search box — it's a small but powerful programming language.

1.3.4 D. Run an inline script

A *script* in QuickNotes is just a sequence of CMline commands. Commands are usually one per line, but they may also be chained on a single line, separated by semicolons. The easiest way to try one is to write it directly into a note as a fenced code block, then run the block in place.

Open a new tab (`Ctrl+N`, *Skip* the title prompt for now) and type this exactly — including the triple-backtick fences and the `qpl` language tag:

````
```qpl
SET greeting = "Hello, world!"
MSG $greeting
```
````

Now position the cursor anywhere inside the fenced block and press **Alt+Enter**. QuickNotes recognizes the `qpl` (or `qns`) language tag, runs the block as a script, and a message box shows `Hello, world!`.

Three ways to run code

Alt+Enter inside a fenced block runs the whole block as a QuickNotes script. Outside a fenced block, it's different: it runs the current line silently – but as a *shell* (operating system) command, not a QuickNotes command. **Ctrl+Enter** also runs the current line as a shell command, and inserts the output below it. The *Tools* menu exposes both, plus a dialog form that lets you type a one-off command without leaving the editor.

1.3.5 5. Bind it to a key

If you find yourself reaching for the same command often, bind it to an available key. Press **F5** and try:

```
LOADKEY MSG Hello!
```

A small dialog pops up asking you to press the key you want to bind it to. Press, for example, **F2**. From now on, any time you press F2 anywhere in the window — text, CMline, doesn't matter — the status line will say `Hello!`. `LISTKEYS` shows your current bindings; `UNLOADKEY <F2>` removes this one.

1.4 What's next?

That was the tour. The rest of the manual breaks each piece down in detail:

- The editor and tabs (chapters 3–4)
- Search and replace, including the interactive `CV` walk-through form (chapter 5)
- The CMline, command chaining, and variable substitution (chapter 6)
- The full Command Reference (chapter 7)
- The QPL scripting language — flow control, variables, functions, the error model (chapter 8)

- User libraries: writing and managing your own `.qnu` files (chapter 9)
- Key bindings, including saving and loading `.qnk` profile files (chapter 10)
- Startup customisation via `startup.qns` (chapter 11)

If you're learning QuickNotes from scratch, read chapters 2 through 6 in order — they build on each other. If you're back to look something up, chapter 7 (Command Reference) and the appendices are the fastest way in.

Chapter 2

The Interface

Chapter 1 walked you through every major feature of QuickNotes at least once. This chapter slows down and gives each one a proper introduction: what's on the screen, what each menu offers, how tabs behave, and where the CMline lives. Most of the parts mentioned here get their own dedicated chapter later, where the keystrokes and command syntax are covered in full — this chapter is orientation.

2.1 The main window

When QuickNotes is running, you'll see (from top to bottom):

- The **title bar** with the QuickNotes brand and, when a note is loaded, the active note's title.
- The **menu bar** with eight menus: File, Edit, Search, View, Tools, Options, Scripts, Help.
- The **tab strip** showing one tab per open note. Unsaved new tabs are labelled `(new)`.
- The **text window** — the main editing area, where you type, select, and move around your note.
- A two-row bottom strip: the **CMline** on top (where you type com-

mands) and the **status line** below it (where QuickNotes reports the result). Each row takes the full width of the window, so long messages on either side stay legible.

Two optional elements appear inside the text window when you enable them: a *line-numbers gutter* (via *Options* → *Line numbers*) and a *Markdown preview pane* alongside the editor (via *View* → *Toggle Markdown preview* or **Ctrl+P**). The preview and PDF export share a compact Markdown parser; Appendix G lists exactly which constructs are supported (and which common ones aren't). A third optional element is the *visible-returns gutter* (via *Options* → *Visible returns* or the `VR` command), a slim right-edge column that shows a pilcrow ¶ at every hard newline so you can tell hard returns from soft word-wraps.

2.2 The menu bar

What follows is a one-line orientation to each menu, with a few keyboard accelerators that are worth knowing right away. The menus carry many more items than these summaries imply; later chapters cover them in detail.

File — new note (Ctrl+N), open a file from disk, save (Ctrl+S), save as… (Ctrl+Shift+S), save the current selection to a new file, close the active note (Ctrl+W), open the `notes/` folder in your operating system's file manager, and exit. All the file operations have CMline equivalents covered in chapter 4.

Edit — undo / redo, cut, copy, paste, select all, plus a *Persistent selection* toggle that keeps a selection alive through any keystroke until you cancel it. The persistent-selection feature gets its own walk-through in chapter 3.

Search — find, find-next, replace, and the interactive step-through replace. The CMline verbs (`SE`, `CH`, `CV`, and their case-aware siblings) are the subject of chapter 5.

View — toggle the Markdown preview pane (**Ctrl+P**).

Tools — a small constellation of ways to run a single line, the current selection, or a one-off command without leaving the editor: as a QPL command, as a shell command with output inserted inline, as a silent QPL line (**Alt+Enter**), and a dialog form that prompts for a command. Chapter 6 covers each in turn.

Scripts — a live-populated list of `.qns` files found in your `scripts/` folder; clicking one runs it directly. *Run script…* (a file picker) and *Open scripts folder* round out the menu. The CMline equivalent is `RUN <name>`, covered in chapter 8.

Options — three independent font dialogs (editor, Markdown preview pane, and a *Styled-PDF* cascade with Font and Margins entries), line numbers, *Visible returns* (pilcrow markers for hard newlines), dark theme, auto-hide menu bar, sync preview scrolling with editor, GUI command line, *Show notes pane* (a toggle that hides the left sidebar so the editor expands leftward), *AutoSave* (suspend or resume the background autosave), and *always on top*. All settings are written to the JSON config file next to `quicknotes.py` automatically.

Help — *Usage / Quick Reference* and *Command Reference* open scrollable in-app reference windows; *About* shows the version and revision banner. The two reference windows are useful while you're learning — they cover much of the same ground as this manual but live inside the program.

2.3 Tabs

Each open note lives in its own tab. The tab strip across the top shows the title of each note (or `(new)` for unsaved ones); click a tab to switch to it. A tab whose note has unsaved changes shows an asterisk in its title.

The smallest useful set of tab operations:

- **Ctrl+N** opens an empty new tab.
- *File* → *Open* loads an existing file from disk into a new tab.

- **Ctrl+W** (or *File* → *Close note*) closes the active tab; if the note has unsaved changes you'll be prompted to save, discard, or cancel.
- **Ctrl+Tab** cycles to the next tab; **Ctrl+Shift+Tab** to the previous.

The CMline has its own verbs for the same operations (`AB` to close, for example), giving you a keystroke-only way to manage tabs from a running script. Chapter 4 covers the full set.

2.4 The text window

The main editing area is a plain Markdown editor. Type whatever you like; the usual Markdown conventions (`#` headings, `**bold**`, `-` lists, fenced code blocks, link syntax) render in the preview pane when you toggle it on.

Day-to-day editing is what you'd expect from any modern text editor: arrow keys and Home / End / Page Up / Page Down for navigation, Shift + arrow for selection, Ctrl + arrow for word-by-word movement, the standard Ctrl+X / C / V / Z / Y for cut / copy / paste / undo / redo. Chapter 3 covers two QuickNotes-specific wrinkles in detail: *insert vs overstrike* typing (toggle with the Insert key) and *persistent selection* mode (a selection that survives any keystroke until you explicitly cancel it with XD / YD).

A note on Markdown

QuickNotes is a Markdown editor in the sense that it understands Markdown syntax and stores files as plain `.md` text; it is not a WYSIWYG editor. You see the literal Markdown source in the editor and a rendered version in the preview pane. If you'd rather work in plain text with no Markdown affordances at all, just don't toggle the preview pane on — the editor doesn't impose Markdown on you.

Nested unordered lists in the preview pane use a three-level bullet cascade that matches the browser default for nested `<ul>`: • (solid round) at the top level, ◦ (hollow round) one level in, and ■ (small square) for any deeper nesting. The styled-PDF export uses the same cascade be-

cause its CSS doesn't override `list-style`, so on-screen preview and printed output agree visually. Wrapped list items align their continuation lines flush with the first-line text, not with the bullet — the wrap indent is computed from the actual bullet width in the current preview font, so it adapts to any UI font / scale.

2.5 The bottom strip: CMline and status line

The bottom of the window has two stacked rows. The **CMline** on top is where you type commands; the **status line** directly below it is where QuickNotes writes its reply. Each row takes the full width of the window, which keeps long file paths, multi-clause error messages, and command echoes legible without truncation.

Pressing **F5** moves focus into the CMline; **F10** toggles between the CMline and the text window. Anything you type and Enter into the CMline is parsed as a *command* — sometimes a search, sometimes a file operation, sometimes a one-line program. The status line then reports what happened: `found`, `no match`, `saved`, the result of a calculation, or an error message.

A few things are worth knowing up-front about the CMline:

- Commands are **case-insensitive**: `se /foo/` and `SE /FOO/` mean the same thing. (The *contents* of search patterns and other arguments may or may not be case-sensitive, depending on the verb — chapter 5.)
- **Tab-completion** works on command names: type `LOA` and press Tab to cycle through `LOADKEY`, `LOADKBD`, `LOADLIB`. Library routine names complete the same way.
- **History** is reachable with the up and down arrows.
- Multiple commands can be chained on one line with `;`. Quote handling, variable substitution, and escape rules are all detailed in chapter 6.

If you want a more prominent command line, *Options → GUI command line* offers the option of a popup dialog: **F5** opens the dialog, **F10** toggles between the CMline and text. The keystrokes and verbs are otherwise identical. The command line at the bottom of the screen is still available for use.

Chapter 3

Editing

This chapter covers the day-to-day mechanics of working in the editor pane: moving the cursor, selecting text, the difference between *insert* and *overstrike* mode, the three keyboard lock keys, and *persistent selection* — a QuickNotes feature that lets a selection survive cursor movement. Each section pairs the familiar keyboard shortcut with its CMline equivalent. The CMline form matters when you start writing scripts; for everyday editing the keyboard is usually enough.

3.1 Cursor movement

The standard movement keys work as you'd expect:

Key	Action
Arrow keys	one character or line
Home / End	start / end of the current line
Page Up / Page Down	one screenful
Ctrl+Left, Ctrl+Right	previous / next word

The CMline equivalents are short two-letter verbs in the XyWrite tradition. The ones you'll reach for most often:

Verb	Cursor target
`TF`	**T**op of **F**ile
`BF`	**B**ottom of **F**ile
`LB` / `LE`	**L**ine **B**eginning / **E**nd
`NW` / `PW`	**N**ext / **P**revious **W**ord
`NS` / `PS`	**N**ext / **P**revious **S**entence
`NP` / `PP`	**N**ext / **P**revious **P**aragraph
`NL` / `PL`	start of **N**ext / **P**revious **L**ine
`PD` / `PU`	**P**age **D**own / **U**p
`JMP n`	**JuMP** to character offset *n*

There are more — `LU` / `LD` (line up / down), `CL` / `CR` (character left or right with line-wrap), `LL` / `LR` (left / right without wrap), `HM` / `BS` (top / bottom of the visible screen), `MU` / `MD` (Move the text Up / Down on screen by one line, XyWrite-style — `MU` reveals more of what's below, `MD` reveals more of what's above; cursor doesn't move). The full set is in Appendix A.

Why two-letter verbs?

QuickNotes inherits XyWrite's preference for terse, mnemonic two-letter command verbs. The pay-off is that you can drive the editor at the speed of typing without ever leaving the home row. TF, BF, NW, PW are faster to type than Top, Bottom, NextWord, PrevWord would be, and the meanings are guessable once you've seen a few.

3.2 Selection

There are three standard ways to select text:

- **Hold Shift while moving the cursor** — Shift combined with any of the movement keys above extends the selection. This is the most common form.
- **Click and drag with the mouse** — click where the selection starts, drag to where it ends.

- **Double-click** selects a word; **triple-click** selects a line; **Ctrl+A** selects the whole document.

The CMline has its own selection verbs, which are most useful inside scripts where you'd rather not generate a synthetic Shift+arrow event:

Verb	Action
`DF`	**D**e**F**ine — start (or end) a selection anchor; subsequent cursor movements extend the selection until you `DF` again or hit `XD`
`DFA`	**D**e**F**ine **A**ll — select the whole document
`DW`	**D**efine **W**ord — select the word at the cursor
`DL`	**D**efine **L**ine — select the current line
`DS`	**D**efine **S**entence
`DP`	**D**efine **P**aragraph
`XD`	Cancel selection (`YD` is a synonym)

The current selection text is available in two ways. The variable `$SE` is substituted into the command line — useful for short, simple selections, like passing a selected filename to a shell command. The expression function `SEL()` returns the same text but as a proper string value, so it survives selections that contain quotes or backslashes:

```
TXT You selected: $SE
SET text = STRIP(SEL())
```

Both are capped at 4 KB; selections longer than that are truncated. But you can save an entire file's worth of text to a variable with `SET myvar = READFILE <filename>`.

3.3 Insert vs overstrike

QuickNotes supports both *insert* mode (typing pushes existing text rightward — the default) and *overstrike* mode (typing replaces what's under the cursor). The **Insert** key toggles between them.

Verb	Effect
`SIN`	**S**et **IN**sert mode
`SOV`	**S**et **OV**erstrike mode
`CIN`	**C**lear **IN**sert (= overstrike)
`TIN`	**T**oggle **IN**sert / overstrike
`TOV`	Toggle (synonym of `TIN`)

The cursor's appearance changes when you flip mode (the I-beam becomes a solid block in overstrike), and the status line confirms the change.

3.4 Lock keys

The three keyboard lock keys — Caps Lock, Num Lock, Scroll Lock — each get a matching set of three CMline verbs:

Lock	Set	Clear	Toggle
Caps Lock	`SCL`	`CCL`	`TCL`
Num Lock	`SNL`	`CNL`	`TNL`
Scroll Lock	`SSL`	`CSL`	`TSL`

The naming follows a simple pattern: **S**et / **C**lear / **T**oggle, then **C**aps / **N**um / **S**croll, then **L**ock. These let a script manage the lock state without relying on the user's current keyboard configuration — handy for routines that want to enter a long ALL-CAPS header or temporarily turn off Num Lock for a numeric-keypad shortcut sequence.

3.5 Persistent selection

Normally a selection is *transient*: any cursor movement that doesn't hold Shift drops it, and typing over it (or pressing Backspace / Delete) replaces or deletes it. In *persistent-selection* mode, the selection sticks until you cancel

it explicitly. You can navigate around the document, edit elsewhere, then return — the selection is still there.

The selection now survives **any** keystroke, not just cursor moves. While it is active, every key behaves exactly as it would if nothing were selected: a printable character is inserted at the cursor, Backspace deletes the one character before the cursor, Delete the one after, and Enter / Tab insert as usual — none of them disturb the highlighted block. The highlight rides along with your edits, so the same characters stay selected wherever the cursor happens to be. Only `XD` / `YD` (or *Edit* → *Cancel selection* / its shortcut) clears it.

Copy and cut are the deliberate exceptions: Ctrl+C and Ctrl+X still act on the highlighted block, since operating on the selection is the whole point of those keys. (Paste, Ctrl+V, replaces the block as usual.)

This is useful when:

- You want to refer back to a selection from several CMline commands without re-selecting each time.
- A script wants to remember a selection across operations that move the cursor.
- You're walking through hits with `SE` and want to operate on each match without losing it.

Toggle persistent selection via *Edit* → *Persistent selection* in the menu, or one of the CMline verbs:

Verb	Effect
`SPS`	**S**et **P**ersistent **S**election
`CPS`	**C**lear …
`TPS`	**T**oggle …

The current state is reported in the special variable `$PS`, which evaluates to `1` (on) or `0` (off). A script that needs persistent selection only temporarily can save and restore the state around its own operations:

```
SET saved_ps = $PS
SPS
# ... operations that rely on persistent selection ...
IF $saved_ps == 0
    CPS
ENDIF
```

3.6 Cut, copy, paste, undo, redo

The standard keystrokes all work as expected:

Key	Action
Ctrl+X	cut
Ctrl+C	copy
Ctrl+V	paste
Ctrl+Z	undo
Ctrl+Y	redo

Most editor actions group into a single undo step, so e.g. a global `CH /foo/bar/` that hits ten matches collapses to one Ctrl+Z. Scripts that do many small edits inside a `DX ... DO` (display-freeze) block also produce a single undo step, so users can roll back a complex transformation cleanly.

Chapter 4

Files and Notes

This chapter is the on-disk view of QuickNotes: where your notes live, what's inside them, when they get saved, and which CMline verbs you'd use to drive any of it from a script. The keyboard shortcuts and menu items were introduced in chapter 2; here we cover the substance behind them.

4.1 The notes folder

QuickNotes stores your notes in a `notes/` folder created next to `quicknotes.py` the first time the program runs. From then on, every saved note lives there as a plain `.md` file, named after the note's title.

You can open a `notes/` file in any other Markdown editor — or `cat` it from a terminal — without going through QuickNotes; they're just text.

The notes pane on the left lists every text-like file it finds in the notes folder, not just `.md` notes. Drop a `.qns` script, a `.qnu` library, a `.json` config, a `.csv`, a `.py` source file into `notes/` and it appears in the pane next to your Markdown notes. The pane is filename-based, so non-`.md` entries display with their extension (e.g. `wrap.qns`, `config.json`); `.md` entries continue to show their YAML title.

4.1.1 Navigating the pane

The pane is designed for fast browsing. Clicking a row or arrowing through with **Up / Down** previews each file *in place*: a single transient tab — the *preview tab* — has its contents replaced row by row as you move. The tab strip stays clean even after browsing through many files. The preview tab behaves like any other tab otherwise — you can scroll it, toggle the Markdown preview pane on it, run CMline commands against it, etc.

To commit a file to a permanent tab so it survives further browsing, press **Return / Enter** or **double-click** the row. The preview tab is "promoted" (it stays open and is no longer the preview), focus moves to the editor, and you can start editing. Subsequent browsing in the pane then creates a fresh preview tab, leaving your committed tab intact.

Useful keyboard handling inside the pane:

Key	Effect
Up / Down / PgUp / PgDn / Home / End	Move through the list; the preview tab's contents update.
Return / Enter	Commit the highlighted row; focus moves to the editor.
Double-click	Same as Return.
Esc	Close the preview tab (if any) and return focus to the editor.
Tab	Standard focus traversal — leaves the pane without changing tabs.

Selecting a file that's *already* open in a permanent tab simply switches to that tab and closes the preview tab (if any). Your committed tabs are never silently clobbered: if you've typed into the preview tab without committing, the next browse "promotes" it automatically rather than discarding the changes.

Find note + browse

The *Find note* search box at the top of the pane and the keyboard browse work well together. Type a few letters into *Find note* to narrow the list, then Tab to the pane and arrow through the matches. Press **Return** when you find the one you want.

4.1.2 Hiding the pane

When you'd rather give the whole window over to the editor, the notes pane can be toggled off entirely. The editor expands leftward to fill the freed space, and toggling the pane back on restores it at its previous width with its current row selection intact.

Trigger	Effect
Options → *Show notes pane*	Menu checkbox; flips visibility.
`SNP` at the CMline	**S**how the **N**otes **P**ane.
`CNP` at the CMline	**C**lear (hide) the notes pane.
`TNP` at the CMline	**T**oggle the notes pane.
`$NP` in an expression	`1` if shown, `0` if hidden.

The visibility state is written to the JSON config file under the `notes_pane_visible` key, so QuickNotes opens with whatever you left it on at last shutdown. A handful of useful idioms:

```
:: Hide the pane while running a wide-screen demo:
CNP

:: Pop it back open and jump cursor to the editor:
SNP; GT

:: Conditional from a key binding -- restore only
if currently hidden:
IF NOT $NP SNP
```

If the notes listbox happens to have keyboard focus when you hide the pane, focus jumps automatically to the active editor so the next keystroke goes somewhere visible.

4.1.3 Other practical points

- **Allow-list of extensions.** The pane is filtered to a curated set of text-like extensions: Markdown, plain text, QuickNotes scripts and libraries, common code (`.py`, `.js`, `.html`, `.css`, `.c`, `.cpp`, `.java`, `.rb`, `.go`, `.rs`, `.sh`, `.bat`, `.ps1`, `.sql`, `.r`, …), data / config (`.json`, `.yaml`, `.toml`, `.ini`, `.csv`, `.tsv`, `.xml`), and a few document formats (`.tex`, `.rst`, `.org`, `.adoc`). Likewise, files with **no extension at all** (`Makefile`, `README`, `LICENSE`, `CHANGELOG`, …) are listed. Binary formats (`.png`, `.pdf`, `.exe`, `.zip`, …) are excluded so a click on the pane never opens a file that would render as gibberish.
- **Dotfiles are hidden.** Names beginning with a `.` (`.gitignore`, `.env`, `.editorconfig`) are skipped regardless of whether their extension is on the allow-list — the rule is there to keep the pane uncluttered by hidden working files.
- **Search includes filenames.** Typing `.qns` into the *Find note* box pulls every script, `wrap` pulls every `wrap.*` file — the search haystack covers the bare filename in addition to the YAML title, the body, and any tags.
- **Saving non-`.md` files preserves their format.** When you save a file the pane opened, QuickNotes emits its YAML frontmatter *only* for `.md` files; everything else is written body-only so that, for example, a `.qns` script stays a `.qns` script.
- **`Ctrl+N` and *New note* still produce `.md` files.** Use *File* → *Open* (or drop the file into `notes/` from your file manager, then refresh) to create or pull in a non-`.md` file. The CMline's `NE <path>` verb also works — it creates whatever extension you give it.

4.2 A note's contents

Every saved note begins with a small YAML-style header recording the note's metadata, followed by the note body. A typical header looks like this:

```
---
title: Things to remember
tags: shopping, meeting
created: 2026-05-11T09:14:00
---

(body of the note starts here)
```

The header is generated automatically when you first save the note; you don't normally need to edit it, but it's plain text so nothing stops you. The fields are:

- `title` — the note's title. Controls both the tab label and the filename. Changing the title in the header (and saving) renames the file on disk.
- `tags` — a comma-separated list. May be empty.
- `created` — an ISO 8601 timestamp, set once on first save and unchanged from then on.

4.3 Tabs

Each open note lives in its own tab. The tab strip at the top of the window is your at-a-glance view of what's open:

- The tab title is the note's title — `(new)` until you save it for the first time.
- A `*` after the title means there are unsaved changes.
- A `[RO]` prefix means the note was opened read-only (see `RE` below).

Keyboard shortcuts for the most common tab operations were listed in chapter 2; here's a short reminder:

Key	Action
Ctrl+N	new (empty) tab
Ctrl+W	close the active tab
Ctrl+Tab	next tab
Ctrl+Shift+Tab	previous tab

4.4 File operations from the CMline

The CMline mirrors every menu file operation, plus a few that have no menu equivalent. These are most useful from inside scripts.

Verb	Action
`NE [path]`	**N**ew note; with a path, create a 0-byte file and open it
`CA <path>`	Open `<path>` in a tab, editable. Aliases: `ED`, `EDIT`
`RE <path>`	Open `<path>` **R**ead-only (`[RO]` prefix). Alias: `READ`
`RW`	Toggle the active tab's **R**ead/**W**rite state. Alias: `READWRITE`
`SA`	**S**ave the **A**ctive tab
`SAAS [path]`	**S**ave **A**ctive tab **AS**; no path opens a dialog
`SAD [path]`	**S**ave selecte**D** text to a new file
`SL`	**S**ave a**L**l open tabs
`SF`	**S**ave and close every open tab (**F**inish)
`AB`	Close the active tab; prompt on unsaved changes (**A**bandon)
`AB/NV`	Close active tab, do **N**ot **V**erify (discard unsaved changes)
`SaveCfg <file>`	Save the full configuration to `<file>` (a `.json` snapshot)
`LoadCfg <file>`	Load a saved configuration snapshot and apply it live

Paths can be absolute, relative, or ~-expanded. Relative paths resolve against the CMline's working directory, which is independent of the `notes/` folder. On Windows, a bare drive letter (`D:`) switches the CMline's working directory to that drive.

A read-only (`[RO]`) tab blocks editing keystrokes with a brief status message and disables autosave for that tab — handy for opening reference files you don't want to modify by accident. `RW` flips the state in either direction.

Where your files actually live

File → Open notes folder in Explorer (or your OS's file manager) reveals the `notes/` folder on disk. For a script-driven version, the special variables `$FI` (the active note's filename) and `$FP` (its full path) give the same information from QPL.

4.5 Autosave

QuickNotes autosaves the active tab quietly in the background after a brief idle period following each edit, so you can't lose more than a few seconds of work even if you forget to Ctrl+S. The asterisk that marks unsaved changes clears the moment autosave catches up.

Autosave is suppressed for `[RO]` tabs.

Autosave is also user-toggleable. *Options → AutoSave* and the CMline verb `AUTOSAVE` (with optional `ON` or `OFF`) both flip the same flag; bare `AUTOSAVE` reports the current state on the status line without changing it, and `$AU` reads it as `"1"` or `"0"`. The setting persists across launches in the JSON config under `autosave_enabled` (default on). When you turn AutoSave **off** mid-session, any in-flight edits remain in the editor buffer — the *write* path is what's suspended, not the buffer itself — and a pending debounced flush is cancelled across every open tab. The buffer is then written to disk only when you ask for it: **Ctrl+S**, the `SA` verb, or the save prompts that fire during `AB/V` (close with verify) and on window close. Closing without saving, or quitting QuickNotes without confirming the prompt, discards the buffered edits — exactly the same trade-off a traditional editor makes.

Chapter 5

Find and Replace

This chapter is about QuickNotes' search and replace facility — the family of CMline verbs that follow XyWrite's terse syntax. The Edit and Search menus drive the same underlying machinery, but the CMline form is where the real power lives: you can combine modifiers, embed escape sequences, chain operations, and use the result as the starting point of a script.

5.1 The three families

QuickNotes splits search-and-replace into three families that share the same argument syntax:

Family	Verbs	What it does
Search	`SE`, `SEB`, `SEA`, `SEBA`	Find the next match; select it
Change	`CH`, `CI`, `CHA`, `CIA`	Replace every match in one pass
Change-Verify	`CV`, `CVA`	Walk match-by-match with Y / N / A / Q

Within each family, the suffix encodes case-sensitivity and (for search) direction. The naming follows a consistent pattern: bare form is case-**I**nsensitive and forward, `A` adds case-**A**bsolute, `B` (search only) reverses direction, and `BA` combines both:

Suffix	Meaning
(none)	case-insensitive, forward
`A`	case-absolute (case-sensitive)
`B`	backward (search only)
`BA`	backward AND case-absolute

`CH` and `CI` are aliases (both case-insensitive replace); `CHA` and `CIA` are likewise aliases (both case-sensitive replace). The `I` / `IA` forms stands for "Change Invisible", a nod to traditional XyWrite usage.

5.2 The self-delimiting argument syntax

Every verb in these families takes its argument in a separator- delimited form. The first character after the verb (and any modifiers — see below) is the *separator*, and the rest of the line is the search term. The separator character must not appear in the search term.

Family	Form
Search	`SE /search term/`
Change	`CH /search/replace/`
Change-Verify	`CV /search#replace/`

Conventionally the separator is `/`, but **any character that isn't already in the search term works**. So if you're searching for a URL that contains slashes, use a different separator:

```
SE !http://example.com/path!
SE ,a/b/c,
SE #fragment#
```

5.3 Modifiers: /W, /T, and /R

Three modifiers may appear after the verb, before the separator. They can be combined in any order:

Modifier	Effect
`/W`	Whole-word match. The pattern is treated as a single word; surrounding characters must be non-word characters.
`/T`	From-Top: forward searches start at line 1; backward searches start at end of file. Either way, the sweep covers the whole file.
`/R`	The search half is a Tcl regular expression. (The replacement half of CH / CV is always substituted literally.)

Examples:

```
SE/W /the/
SE/T /Introduction/
SEA/W/T /TODO/
CH/W /color/colour/
SE/R /\d{3}-\d{4}/
CH/R /\n\n+/\n\n/
```

The first finds `the` as a complete word, skipping `there` and `father`. The second starts from the top of the file rather than the cursor. The third combines case-sensitive, whole-word, and from-top. The fourth replaces `color` with `colour` only as a whole word, so `colorblind` is left untouched. The fifth finds a US-style local phone number with a regex. The sixth collapses any run of two or more blank lines down to one, in a single pass.

`/W` uses Tcl's regex word-boundary anchors internally; metacharacters in your pattern are escaped automatically, so `SE/W /a.b/` looks for the literal three-character string `a.b` (dot included), not "a, any character, b."

5.4 The /R (regex) modifier

With `/R`, the search half of the verb is handed to Tk's regex engine (Tcl-flavor regular expressions) instead of being matched literally. The full Tcl regex grammar is documented in the Tcl re_syntax man page; the most common features are:

Construct	Meaning
`.`	Any single character (not newline by default)
`* + ?`	Zero-or-more, one-or-more, zero-or-one
`{m,n}`	Repetition count
`[abc] [^a]`	Character class / negated class
`\d \w \s`	Digit, word-char, whitespace
`\m \M`	Word-boundary anchors (start / end of word)
`^ $`	Start / end of line
`(...)`	Grouping (capturing, but see caveat below)
`|`	Alternation

`/R` combines with the other modifiers:

Combination	Behavior
`/R`	Plain regex search
`/R` + `/W`	Regex search, wrapped in `\m...\M` so the match must occupy a whole-word span
`/R` + `/T`	Regex search, starting at the top (or end) of the file
`/R` + case-absolute verb (SEA, CHA, CIA, CVA)	Match is case-sensitive

Backreferences in the replacement. With `/R` in effect, the replacement half recognizes these tokens:

Token	Meaning
\1 … \9	The text matched by capture group N. Groups are numbered left-to-right by opening parenthesis. References to groups that didn't match or don't exist expand to the empty string.
&	The whole match.
\&	A literal & (use this when your replacement actually wants an ampersand).
\X (other)	The character X, with the leading backslash dropped. So \: is just :.

Example — swap "Last, First" name pairs to "First Last":

```
CH/R "(\w+),\s{0,}(\w+)"\2 \1"
```

The pattern captures two word groups; the replacement reassembles them in the opposite order with a space between. Without /R you'd need a script to do this; with /R it's one line. Note the choice of " as the separator: / would be hard to read among the regex backslashes, and " doesn't collide with any regex metacharacter, so the separator stands out cleanly from the pattern.

Example — wrap every bare website address in angle brackets:

```
CH/R "https?://\S+"<&>"
```

In the replacement <&>, the & stands in for whatever the pattern matched; the angle brackets are literal. The / characters inside https:// would break a /-delimited form here (six slashes, not three) — another reason to reach for " when your pattern is likely to contain /.

Bad regex is a trappable error. A malformed pattern (unbalanced parentheses, bad repetition count, etc.) sets $ERROR and reports bad regex: <message> on the status line. The text is not modified.

Escape processing happens first. Both halves of the argument are run through the QPL string-literal escape table before the regex engine sees them. So \n and \t mean a real newline and tab in the pattern just as they do without /R. To pass a backslash through to the regex engine, double it: CH/R

`/\\d+/N/` matches one or more digits, because the QPL escape pass turns `\\d` into `\d` and the regex engine then interprets that as the digit class.

5.5 Escape sequences in patterns and replacements

Both halves of a search-or-replace argument honor the standard QPL string-literal escape sequences:

```
SE /\n/
CH /\t/    /
CH /\n\n\n/\n\n/
SE /\xa0/
```

In order: find a real newline; replace each tab with four spaces; collapse runs of exactly three blank lines down to two (run this a few times to flatten longer runs); find a non-breaking space.

The recognized escapes are `\n`, `\t`, `\r`, `\0`, `\\`, `\"`, `\xHH`, `\dNNN`, `\uHHHH`; unrecognized escapes pass through verbatim. To embed a literal backslash followed by `n` (i.e. the two characters `\` and `n`), double the backslash: `\\n`. Windows paths inside a search pattern are safest passed via a `$`-variable set from a `SET = "..."` expression, since the SET substitutor doubles backslashes inside `"..."` so the round-trip preserves them.

5.6 After-match behavior

Search verbs leave the matched text selected and the cursor positioned at the end (forward search) or start (backward search) of the match. A repeated `SE` continues in the same direction from there, so `SE /TODO/` followed by another `SE /TODO/` walks the TODOs one at a time. The status line reports `found` on success or `not found` on failure, which also sets `$ERROR`.

Search is a *single sweep* in the direction of the verb — cursor to end-of-file (forward) or cursor to top-of-file (backward) — and does NOT wrap around. This is how XyWrite behaves and it's what loop-with-`$ERROR` scripts depend on: a script that repeatedly calls `SE` until `$ERROR` is set walks every match in the file exactly once and then exits, even if there were matches before the starting cursor position. Use `/T` to force the sweep to cover the whole file from the top.

Change verbs report the count: `CH: replaced 12 occurrences`. A zero-count replace is a trappable failure — `$ERROR` is set and the status line reads `not found: <pattern>`. The whole pass collapses to a single Ctrl+Z step, so a runaway replace can be undone in one keystroke.

Change-Verify (`CV` / `CVA`) walks each match in turn, highlights it, and pops a dialog with four options:

Key	Action
Y	replace this one; advance to the next
N	skip this one; advance to the next
A	replace this one and all remaining (no more prompts)
Q	quit; leave the rest unchanged

Esc is treated as Quit. The whole CV pass — replacements made via Y and via A — collapses to a single undo step, exactly like CH.

5.7 Worked examples

Step through a list of TODOs. Land on the first one, then hit **F9** repeatedly to walk forward through each match. Use `XD` to cancel the selection if a TODO is found (note that the IF statement works from the CMline):

```
SE /TODO/; IF NOT $ERROR XD
```

Replace tabs with four spaces, document-wide. The four spaces go between the second and third `/`:

```
CH/T /\t/    /
```

Selectively rename a variable in a script. Case-absolute matters here — you don't want `Old_Name` (a class name) or `OLD_NAME` (a constant) silently renamed. The `CVA` walks you through each match with the usual Y / N / A / Q prompts:

```
CVA/T /old_name/new_name/
```

Collapse multi-blank-line runs to single blank lines. With `/R` this is a one-shot replace; the regex `\n\n+` matches two-or-more newlines and the literal replacement puts back exactly two (one trailing newline + one blank line + the next paragraph's leading text):

```
CH/T/R /\n\n+/\n\n/
```

Without `/R` you'd need a loop, because a literal pattern can only fold one extra blank line per pass:

```
WHILE 1
    CH/T /\n\n\n/\n\n/
    IF $ERROR BREAK
ENDWHILE
```

Re-using a selection in the search

The special variable `$SE` holds the current selection. So after selecting a word with DW, `SE /$SE/` searches for the next occurrence of that same word. The expression function `SEL()` returns the same value but as a proper string for contexts where a literal substitution would not work.

Chapter 6

The Command Line (CMline)

The CMline is the one-line command bar at the bottom of the QuickNotes window. Most things you'd do via a menu, plus a great deal you can't, are reachable as a short CMline verb. Chapter 1 introduced it; chapters 3 through 5 reached for it whenever they needed to drive the editor programmatically. This chapter is the CMline itself, end to end.

6.1 Where it lives, how to enter it

The CMline strip sits across the bottom of the window, with the status line underneath. When the CMline has focus, a cursor blinks inside it ready for your input.

Focus moves like this:

Key	What it does
F5	Jump to the CMline and clear it.
F9	Execute the command, if any, on the CMline.
F10	Toggle focus between CMline and editor.

Commands can be executed with **Enter** as well as **F9**.

6.2 The GUI command-line option

For longer commands, the bottom-strip CMline can feel cramped. The *Options → Use GUI command line* toggle replaces it with a popup dialog above the editor. Tab-completion, history, escape rules, and every other CMline behavior described in this chapter work identically inside the dialog.

Keys inside the GUI dialog:

Key	Effect
Enter / F9	Run the command; hide the dialog (text preserved on the hidden popup).
F10	Hide the dialog without running; in-progress text is preserved.
F5	Clear the entry (start fresh).
Esc	Close (destroy) the dialog; in-progress text is discarded.
Up / Down	History recall.
Tab	Verb completion.

Keys from the editor when the GUI option is on:

Key	Effect
F10	Re-open the dialog. If the dialog already holds text (preserved from a previous hide, or staged via `BC`), the text reappears ready for editing or re-execution.
F5	Re-open the dialog **cleared** for a fresh command.
F9	Execute the bottom-strip CMline's content directly — no GUI dialog opens.

When a GUI-dialog **Enter** runs a command, the executed text is mirrored into the bottom-strip CMline (and into its history) so **F9** from anywhere re-executes it without re-opening the dialog. The two surfaces share a single canonical text stream (`self._cmdline_var` internally; exposed via the `BC` verb below); the GUI dialog is a transient editor for that stream.

The option is written to the config file (`gui_cmdline`), so you can leave it on or off as you prefer. Three CMline verbs let scripts and key bindings drive the toggle without touching the menu:

Verb	Effect
`SGC`	**S**et **G**UI command line ON.
`CGC`	**C**lear (off) **G**UI command line; tears down any visible dialog.
`TGC`	**T**oggle **G**UI command line.

The current state is reported by the read-only special variable `$GC` (1 when on, 0 when off).

6.3 Staging and executing: `BC` and `XC`

Two verbs let scripts and key bindings drive the CMline itself the way a XyWriter would expect.

`BC [<text>]` — **B**ottom **C**Mline. Behaves exactly like pressing **F5**: it blanks the command line, summons the GUI CMline popup if that option is enabled, and moves typing focus to the command line. If `<text>` follows, that text is first staged into the command line — ready to edit or to F9-execute — and left selected; a bare `BC` leaves the line blank. The rest of the line after `BC` is taken literally (no shlex tokenization), so paths, separators, embedded spaces, and `$NAME` substitutions all survive verbatim.

Unlike older versions, `BC` now summons and focuses the command line in *every* context, including from inside a running script — it is a deliberate "jump to the command line" gesture wherever it is issued. The staged text is still written to the bottom-strip CMline, so it remains the source of truth for an immediately following `XC` and for F9 re-execution.

`XC` — e**X**ecute **C**Mline. Dispatches whatever is currently on the bottom-strip CMline, exactly as if you'd pressed F9. Takes no arguments. Refuses to recurse if the CMline content is itself `XC`.

Together they form the classic *stage-and-fire* idiom:

```
BC SE/T /TODO/; XC
```

That single line builds the command `SE/T /TODO/` in the CMline, then runs it. A subsequent F9 (or another `XC`) repeats the run. A subsequent F10 (with the GUI option on) brings the same text up in the GUI dialog for tweaking.

6.4 Running a command

You type a command, press Enter, and the status line shows what happened. Most commands are case-insensitive: `se /foo/` and `SE /FOO/` mean the same thing, although the *contents* of the search pattern may or may not be case-sensitive depending on which verb you used (see chapter 5).

The smallest useful CMline session:

```
PR Hello
```

prints `Hello` to the status line. It's the QuickNotes equivalent of `echo`, and chapter 8 will return to it many times as a quick way to inspect script state.

6.5 Chaining: the semicolon

Multiple commands on one line are separated by `;`. They run strictly left-to-right; if one fails, the next still runs (unless that next command tests `$ERROR`):

```
SE /TODO/; PR Found one
```

Two backslash escapes are honored inside a chained line, and only inside chained lines:

- `\;` is a literal semicolon (NOT a separator).

- `\\` is a literal backslash.

So `MSG hello\; world` shows the message `hello; world`. A lone `\` not followed by `;` or another `\` is preserved as-is, so Windows paths don't need quoting:

```
RE C:\notes\meeting-2026.md
```

A few verbs are exempt from the semicolon splitter — `X`, `XY`, `LOADKEY`, and the `SE` / `CH` / `CV` search-and-change family — because their arguments naturally contain things that look like separators or escapes. The splitter recognizes these by name and hands them the rest of the line verbatim.

6.6 Variable substitution

Anywhere a `$NAME` appears in a command (case-insensitive), it's replaced with the value of the variable at dispatch time:

```
SET who = world
PR Hello, $who!
```

prints `Hello, world!`. Names are alphanumeric (plus underscore); the first character is a letter or underscore. A literal `$` that shouldn't be substituted is written `\$`:

```
PR Price: \$5.99
```

QuickNotes also gives you a long list of *special variables* — read-only values that reflect the program's current state, like `$SE` (current selection), `$FI` (active note's filename), `$FP` (full path), `$ERROR` (last-command error flag), `$OS` (host OS). The full set is in Appendix B; here are the ones you'll reach for most often:

Variable	Value
`$SE`	current selection (capped at 4 KB)
`$ERROR`	`1` if the last error-aware command failed, else `0`

Variable	Value
`$FI`	active note's filename (without path)
`$FP`	active note's full path
`$OS`	host OS: `Windows`, `Darwin`, `Linux`
`$RV`	sub/library routine return value
`$RC`	last shell command's exit code

Arrays (set by `SPLIT` / `SPLITS`) are accessed with bracket sugar: `$parts[0]` is the count, `$parts[1]` through `$parts[N]` are the elements. See chapters 5 and 8 for examples.

6.7 Comments

A # followed by whitespace (or end-of-line) starts a comment that runs to the end of the line. Comments are stripped before dispatch, so they're invisible to the splitter and the rest of the line:

```
SE /TODO/ ; PR Found one    # remind myself
```

The whitespace context rule prevents # from triggering a comment in everyday content — `Issue #4 is fixed`, `#hashtag`, `https://x.com#frag` all pass through unchanged. If you really need a literal # with whitespace around it, escape it: `\#`.

Inside `"..."` string literals, # is always literal regardless of whitespace context. And `X` / `XY` pass the rest of the line to the shell verbatim, so a # there is interpreted by the shell, not by QuickNotes.

6.8 History

Pressing **Up** in the CMline walks back through your previous commands; **Down** walks forward again. The history holds the last 100 distinct commands. When a command is run, any prior occurrence of the same line is

removed from the history before the new entry is appended at the end, so each command appears exactly once with the most-recently-used at the bottom. That means an Up-arrow walk never asks you to step past dozens of copies of `SE /todo/` to reach the previous distinct command — it shows you a hundred different things you've actually done. History persists across sessions, so the command you ran yesterday is still reachable today.

The `HISTORY` verb pops a popup that lists the history (optionally filtered by a number or substring; see Appendix A). Inside the popup, **Up / Down** navigate, **Return** executes the highlighted line, **Del** removes it from the history (the trim persists at the next save of the config file), and **Esc** closes the popup without executing anything.

6.9 Tab-completion

Pressing **Tab** with the cursor at the end of a partial verb name cycles through every command and library routine that starts with what you've typed. So `LOA` + Tab cycles through `LOADKEY`, `LOADKBD`, `LOADLIB`; `MS` + Tab gives you `MSG`.

Library routines (chapter 9) complete the same way, so if your `user.qnu` defines a routine called `STAMP`, then `ST` + Tab will offer it alongside the built-in `SUB`, `SET`, etc.

Tab in the *middle* of a partial verb behaves the same as Tab at the end — completion works from cursor-position backward to find the start of the word.

6.10 Shell-out: X, XY, XV

Three verbs hand the rest of the line off to the operating system's shell:

Verb	Effect
`X <cmd>`	run `<cmd>`; status line shows exit code

Verb	Effect
`X/NV <cmd>`	launch `<cmd>` and return immediately (no wait, no output capture)
`XY <cmd>`	run `<cmd>`; **Y**ank stdout into the buffer at the cursor
`XV <var> <cmd>`	run `<cmd>`; capture stdout into `$<var>`

These are the bridge between QuickNotes and the rest of your toolbox. `XV` is the workhorse — once a shell command's output is in a variable, the rest of QPL can work with it. Examples:

```
XV today date /T
PR Today is $today.
```

That's Windows. On Linux or macOS, where the shell handles `date` a little differently:

```
XV today date +%Y-%m-%d
PR Today is $today.
```

A version of the same idea that works on all three platforms because Git's CLI is consistent:

```
XV branch git branch --show-current
WR Working on branch: $branch
```

For scripts that need to run cross-platform, the special variable `$OS` lets you branch on the host: `$OS == "Windows"` vs `Linux` vs `Darwin`. Chapter 8 has an example using `$OS` in an `IF`.

`X` and `XY` consume the whole rest of the line as the shell command — no chaining or `$`-substitution past the verb. `XV` takes its varname as a single token first, then everything after that is the shell command. The shell command's exit code lives in `$RC` afterward, so a script can branch on it.

`X`, `XY`, and `XV` run in a background worker thread, but the script runner waits for them at end-of-line: the next script line doesn't begin until `XY`'s

output has been inserted, `XV`'s variable has been populated, and `$RC` reflects the final exit code. So a script can write `XY echo hello` on one line and `TF; LD; DL` on the next, confident that the second line sees the buffer the first line built. (For true fire-and-forget where the script proceeds immediately, use `X/NV`.)

6.11 Working directory

The CMline tracks its own current working directory, separate from your `notes/` folder. Relative paths in `CA`, `RE`, `XV`, etc. resolve against this CMline cwd. The bare drive-letter form `D:` (Windows only) switches the cwd to the root of that drive. Chapter 4 covered the file-tab side of this; the point worth making here is that the CMline cwd is an integral part of how file-taking verbs behave.

6.12 Putting it together

Here's a CMline session that uses most of the pieces in this chapter at once. We'll find every TODO in the current note, mark it in the status line as we go, count them, and pop a summary dialog:

```
SET n = 0
:loop
    SE /TODO/
    IF $ERROR ; GOTO :done ; ENDIF
    SET n = $n + 1
    GOTO :loop
:done
MSG Found $n TODOs.
```

Save that as `count-todos.qns` in your scripts folder and you can fire it from the CMline with `RUN count-todos` (or from the Scripts menu). Chapter 8 dives into the QPL machinery that makes this possible.

Chapter 7

Command Reference

This chapter gives a lookup view of every command-line verb QuickNotes recognizes, grouped by what each one does rather than alphabetically. For an alphabetical index, see Appendix A.

Entries follow a consistent pattern:

> **VERB** *arg form* — one-line description.

Verbs that were introduced and explained in earlier chapters get a brief entry here with a cross-reference back to the chapter that covers them in full. New verbs get a short example where one helps. The control-flow verbs that make up the QPL scripting language proper (`SET`, `IF`, `WHILE`, `GOTO`, `CALL`, `RETURN`, `SUB`, `LOCAL`, `QUIET`, `CLERR`, `SETERR`, `RUN`, `RUNHERE`) are listed for completeness at the end of the chapter; chapter 8 covers their syntax and behavior in depth.

7.1 Files and tabs

Covered in detail in chapter 4.

Form	Description
NE [path]	New note; with a path, create a 0-byte file and open it.
CA <path>	Open <path> in a tab, editable.
RE <path>	Open <path> read-only ([RO] prefix).
RW	Toggle the active tab's read/write state.
SA	Save the active tab.
SAAS [path]	Save as; no path opens a dialog.
SAD [path]	Save selected text to a new file.
SL	Save all open tabs.
SF	Save and close every open tab.
SaveCfg <file>	Save the full configuration to <file> (.json snapshot; prompts before overwrite).
LoadCfg <file>	Load a configuration snapshot and apply it live.
AB	Close active tab without verifying. (Alias: AB/NV.)
AB/V	Close active tab; if modified, prompt save / discard / cancel.
ABD	Delete the active note from disk and close its tab; no confirmation.
REC, RECALL	Discard edits since the last manual save and revert the active tab to that snapshot.
NX / PX	Next / previous tab.
D:	Switch the CMline working directory to drive D (Win).

Aliases: CA = ED = EDIT; RE = READ; RW = READWRITE.

7.2 Cursor movement

Covered in chapter 3. Two-letter verbs in the XyWrite tradition.

Verb	Cursor target
TF	Top of file
BF	Bottom of file
LB / LE	Line beginning / end
NW / PW	Next / previous word
NS / PS	Next / previous sentence
NP / PP	Next / previous paragraph

Verb	Cursor target
`NL` / `PL`	Start of next / previous line
`LU` / `LD`	Line up / down
`CL` / `CR`	Cursor left / right (wraps to adjacent line)
`LL` / `LR`	Left / right without wrap
`NC` / `PC`	Next / previous character (synonyms for `CR` / `CL`)
`PD` / `PU`	Page down / up
`MU` / `MD`	Move text up / down on screen by one line (XyWrite-style: `MU` reveals the line below, `MD` reveals the line above); cursor doesn't move
`HM` / `BS`	Cursor to top / bottom of visible screen
`JMP n`	Jump to character offset *n* (0-based)

7.3 Selection

Covered in chapter 3.

Verb	Action
`DF`	Define — start (or end) a selection anchor
`DFA`	Define all — select the whole document
`DW`	Define word
`DWONLY`	Like `DW` but always single-word
`DL`	Define line
`DS`	Define sentence
`DP`	Define paragraph
`XD`, `YD`	Cancel selection
`CP`	Copy the selected text to the cursor position
`MV`	Move the selected text to the cursor position (delete from origin, insert at cursor)

7.4 Deletion

Verb	Action
`DN`	Delete the current selection (or character)
`BD`	Backspace — delete the character before the cursor

Verb	Action
`RC`	Rub out — delete the character under (right of) the cursor
`RD`	Delete selected text
`RL`	Delete the rest of the line (from cursor to end)
`RS`	Delete the current sentence

7.5 Case

Each of the three verbs below acts on the current selection. If there is no selection, the verb acts on the single character under the cursor instead.

Verb	Effect
`LC`	Lowercase.
`UC`	Uppercase.
`CC`	Swap case (each character's case is flipped individually, so `Hello` becomes `hELLO`).

7.6 Insert vs overstrike

Covered in chapter 3.

Verb	Effect
`SIN`	Set insert mode
`SOV`	Set overstrike mode
`CIN`	Clear insert (= overstrike)
`TIN`, `TOV`	Toggle

7.7 Lock keys

Covered in chapter 3.

Lock	Set	Clear	Toggle
Caps Lock	`SCL`	`CCL`	`TCL`
Num Lock	`SNL`	`CNL`	`TNL`
Scroll Lock	`SSL`	`CSL`	`TSL`

7.8 Persistent selection

Covered in chapter 3.

Verb	Effect
`SPS`	Set persistent selection
`CPS`	Clear persistent selection
`TPS`	Toggle

7.9 Menu-bar visibility

Mirror the Alt-toggle hotkey from CMline and scripts. Covered in chapter 11.

Verb	Effect
`SMB`	**S**how the menu bar.
`CMB`	**C**lear (hide) the menu bar.
`TMB`	**T**oggle menu-bar visibility.

These change only the *current* state; they don't flip the persistent `auto_hide_menu_bar` config key.

7.10 Notes-pane visibility

Mirror the *Options* → *Show notes pane* checkbox from the CMline and scripts. Covered in chapter 4. Read `$NP` to test the current state (`1` shown, `0` hidden).

Verb	Effect
SNP	**S**how the **N**otes **P**ane.
CNP	**C**lear (hide) the notes pane.
TNP	**T**oggle the notes pane.

When hidden, the editor expands leftward to fill the freed space. The sidebar's listbox state and current selection survive the hide / show cycle, so toggling back on is instant. The choice persists across launches via the `notes_pane_visible` config key.

7.11 Search and replace

Covered in chapter 5.

Form	Description
`SE /pattern/`	Find a pattern.
`CH /search/replace/`	Replace every match.
`CV /search/replace/`	Walk match-by-match with a Y / N / A / Q prompt.

Suffix `B` = backward, `A` = case-absolute. So `SEB` is backward search, `SEA` is case-absolute forward, `SEBA` is both. `CH` / `CHA` and `CI` / `CIA` are aliases. Modifiers `/W` (whole-word), `/T` (from top), and `/R` (regex) may appear after the verb in any order, e.g. `SE/W/T /TODO/` or `CH/R /\d+/N/`.

7.12 Insertion (buffer text I/O)

Form	Description
`WR <text>`	Insert `<text>` at the cursor; escape sequences (`\n`, `\t`, …) honored.
`WRL <text>`	Like `WR`, but appends a trailing newline.

Aliases: `WR` = `WRITE`; `WRL` = `WRITELN`.

Example:

```
WR Updated:\t\d048\d050\d050\d055
```

inserts `Updated:` followed by a TAB and the ASCII characters `0`, `2`, `2`, `7` (decimal `048`, `050`, `050`, `055`).

7.13 File I/O

Form	Description
`RF <path> <var>`	Read the file's contents into `$<var>`.
`WF <var> <path>`	Write `$<var>` to `<path>` (overwriting).

Aliases: `RF` = `READFILE`; `WF` = `WRITEFILE`.

7.14 Clipboard

Verb	Effect
`CLIP <text>`	Replace the clipboard with `<text>` (the literal rest of the line). Bare `CLIP` clears the clipboard.
`CY <text>`	Alias for `CLIP`.
`APPEND`	Append the current selection to the clipboard's existing contents.
`PREPEND`	Prepend the current selection to the clipboard's existing contents.

`APPEND` and `PREPEND` leave the selection itself unchanged; only the clipboard is modified. They build up a multi-piece clipboard from several places in the document, ready for one `Ctrl+V` later. Both set `$ERROR` if no selection is active.

`CLIP` / `CY` take the rest of the line as the literal payload (no shlex tokenization, so spaces and punctuation pass through verbatim). `$`-substitution applies first, then string-literal escape sequences (`\n`, `\t`, `\xHH`, `\uHHHH`, …) are processed — matching `PR` / `MSG` / `WRITE` conventions. Double a backslash to embed a literal one. Examples:

```
CLIP Hello, World!
PR $CB                          ; "Hello, World!"

SET path = "C:\\Users\\you\\notes"
CLIP $path
PR $CB                  ; the literal Windows path

CLIP First line\nSecond line
PR $CB                      ; two-line clipboard

CLIP                       ; clears the clipboard
```

A common idiom: `CLIP` once to set a baseline, then `APPEND` / `PREPEND` to build up combined content from selections in the document.

7.15 Counts

Verb	Action
`CHARCOUNT`	Status-line report of the character count of the current selection (or, if no selection, the whole document).
`WORDCOUNT`	Same, but word count.

7.16 Splits

Covered in chapter 8 (alongside the variable / array machinery).

Form	Description
`SPLIT <name> <input> <delim>`	Split `<input>` around `<delim>` (literal substring); parts in `$<name>_0..$<name>_N`.
`SPLITS <name> <input> <set>`	Like `SPLIT` but `<set>` is a set of delimiter characters.

Both honor string-literal escapes in `<input>` and `<delim>`, so `SPLIT lines "$text" "\n"` splits on real newlines.

7.17 Information dialogs

Form	Description
`PR <text>`	Print `<text>` to the status line.
`MSG [/T:N] <text>`	Modal info dialog; `/T:N` auto-dismisses after N seconds.
`TXT <text>`	Scrollable monospace dialog (for multi-line / pre-formatted output).
`INPUT <var> [prompt]`	Modal entry dialog into `$<var>`; sets `$ERROR` on cancel.

All four honor string-literal escapes (`\n`, `\t`, `\\`, …).

7.18 Environment & existence

Form	Description
`ENV <name> <var>`	Copy environment variable `<name>` into `$<var>` (empty string if unset).
`EXISTS <path> <var>`	Store `1` or `0` in `$<var>` depending on whether `<path>` exists.

7.19 Display freeze

Verb	Effect
DX	Display off: suppress screen refreshes
DO	Display on: cancel DX and flush pending updates

Useful for scripts that do many small edits and don't want the screen flickering. DO fires automatically at end-of-script if the script forgot.

7.20 User Interface (UI) size

Verb	Effect
UIFONT [<n>]	Set the UI font size (menus, tabs, sidebar, dialogs without explicit fonts) to <n> points. With no argument, reports the current size. Live re-apply.
EDFONT [<n>]	Set the editor font size (the text-editing area) to <n> points, between 6 and 72. With no argument, reports the current size. Live re-apply. The editor font *family* and *style* are unchanged — use the editor Font dialog for those.
UISCALE [<f>]	Set the global Tk scaling factor. Affects spacing, padding, scrollbar widths, etc. in addition to text. Restart QuickNotes for the full effect.
UIRESET	Restore both to their factory defaults (12 pt font, 1.5 scale).
OF [<n>]	Set the editor / preview margin offset to <n> character columns of the editor font (symmetric: left = right = <n>). With no argument, reports the current value. Negatives clamp to 0.
LEADING [<n>], LSP [<n>]	Set the editor / preview line spacing (leading) to <n> percent of the editor font's natural line height. 0 = the Tk default (no extra leading). With no argument, reports the current value. Clamped to 0..200. Applies to both the editor and the in-app Markdown preview; the pixel gap re-scales when the font size changes. Also reachable from *Options → Line spacing…*

Both UIFONT and UISCALE values persist to the config file (ui_font_size and ui_scale keys) and survive across sessions. They're also exposed read-only as $UIFONT and $UISCALE. The editor font size set by EDFONT persists under the font_size key

(shared with the editor Font dialog) and is exposed read-only as `$EDFONT`. The margin offset has its own special variable `$OF`. The line-spacing percent persists under `editor_line_spacing_pct`.

`EDFONT` is the editor-text counterpart to `UIFONT`: where `UIFONT` sizes the chrome around the page, `EDFONT` sizes the text on it. Because tab stops, the margin offset, and line leading are all measured in the editor font, changing `EDFONT` re-tracks those automatically.

The same two knobs are reachable from **Options → UI font size** and **Options → UI scale**, with a handful of preset sizes pre- defined for one-click change.

A quick tour of why two knobs:

- `UIFONT` adjusts the *text* in UI chrome: menu labels, menu pulldown items, tab names, the "Find note" entry, the note list, dialog buttons. Larger text without disproportionately enlarging the surrounding padding. Live re-apply means a change takes effect on the next redraw without restarting.
- `UISCALE` adjusts the *whole metric system* Tk uses internally. A 1.5 scale makes a 9-pt font draw at the pixel size you'd normally see from a 13.5-pt font *and* expands borders, gaps between widgets, the height of the menu bar's clickable area, and so on, in proportion. Tk caches some metrics at widget- creation time, so a `UISCALE` change requires a restart to fully take effect.

Dialog windows (Find & Replace, the three Font dialogs, Press-a-key, Usage, Command Reference, etc.) auto-scale their pixel dimensions against the current `UIFONT` and `UISCALE` settings, so the buttons at the bottom of a dialog remain visible at every size. Newly-created dialogs read the current settings; existing dialogs use whatever was in effect when they were opened.

For aging eyes, the combination of `UIFONT 12` and `UISCALE 1.5` (both defaults) produces an interface that's noticeably easier to read than the Windows 9-pt-at-1.33 baseline without looking oversized. For higher-

density screens (4K monitors at native resolution), users may want `UIFONT 14` or `UISCALE 1.75`.

7.21 Shell-out

Covered in chapter 6.

Form	Description
`X <cmd>`	Run `<cmd>` in the shell; exit code in `$RC`.
`X/NV <cmd>`	Launch `<cmd>` in the shell and return control to the QPL script immediately; stdout / stderr are discarded. `$RC` is `0` on a successful launch, `-1` on launch failure. Useful for starting a background process and then polling (e.g. with `EXISTS` + `WAIT`) for some side-effect such as an output file.
`XY <cmd>`	Run `<cmd>`; yank stdout into the buffer at the cursor.
`XV <var> <cmd>`	Run `<cmd>`; capture stdout into `$<var>`.

7.22 Sorting

`SORT [/W] [/R] [/D] [/F:n] [/N:n]` sorts the lines of the current selection. Modifiers, combinable in any order:

Modifier	Effect
`/W`	Word mode — use the first word of each line as the sort key.
`/R`	Reverse (descending).
`/D`	Dedup — drop adjacent duplicates after sorting.
`/F:n`	Skip the first *n* characters of each line when building the sort key.
`/N:n`	Use only the next *n* characters (after any `/F:n` skip) as the sort key.

Example:

```
SORT/D
```

sorts the selected lines and removes adjacent duplicates.

7.23 Undo / redo

Form	Description
`UNDO [n]`	Undo the last change. With `n`, undo up to *n* steps.
`REDO [n]`	Redo the last undone change. With `n`, redo up to *n* steps.

Both stop early if there's nothing left on the corresponding stack and report the number of steps actually performed in the status line.

7.24 Focus

Verb	Effect
`GH`	Go to CMline (focus the command line, no clearing)
`GT`	Go to text (focus the editor pane)

7.25 Help and discovery

Verb	Effect
`?`	About dialog
`ABOUT`	Same as `?`
`HELP`	Open the in-app Usage / Quick Reference window
`COMMANDS`	Open the in-app Command Reference window
`LISTLIB`	List loaded library routines (chapter 9)
`LISTKEYS`	List current key bindings (chapter 10)
`LISTVAR`	List script variables currently in memory (chapter 8)
`QUIT`	Close the QuickNotes window; prompts per modified tab
`QUIT/NV`	Close the QuickNotes window without prompting; unsaved changes discarded

7.26 Filesystem

Commands for working with files on disk from inside QuickNotes, independent of which file you have open in the editor.

Form	Description
`CD [path]`	Show or change the CMline working directory. Bare `CD` reports the current value.
`CD/D <drive>:<path>`	Windows-only: change drive and directory in one step.
`COPY <src> <dst>`	Copy a file on disk. Prompts before overwriting an existing destination.
`COPY/NV <src> <dst>`	Like `COPY`, but overwrites the destination silently.
`DEL <path>`	Delete a file on disk. Prompts to confirm.
`DEL/NV <path>`, `ERNV <path>`	Like `DEL`, but skips the confirmation prompt.
`RENAME <old> <new>`, `REN <old> <new>`	Rename `<old>` to `<new>` in the same directory. `<new>` must be a bare filename (no path separators) — use `MV`-style approaches via `COPY` + `DEL` for cross-directory moves. Refuses directories. Prompts before overwriting an existing destination. If `<old>` is open in a tab, the tab follows the rename in place.
`RENAME/NV <old> <new>`, `REN/NV <old> <new>`	Like `RENAME`, but overwrites an existing destination silently.
`DIR [path] [/switches]`	Scrollable directory listing. Accepts a curated subset of Windows `DIR` switches (`/A`, `/O:N`, `/B`, …). Up/Down to navigate; type to find; F3 finds again; Enter (or double-click) opens the highlighted file in a new tab; Ctrl+Enter opens it in its associated OS application or launches it; a bare Ctrl tap (press and release with no other key) copies the highlighted entry's full path to the clipboard; Del deletes the highlighted file (with confirmation).
`LS [path] [/switches]`	Alias for `DIR`.

Form	Description
`MERGE <path>`, `ME <path>`	Insert the contents of `<path>` into the active note at the cursor (insert mark). Replaces the current selection, if any.
`MERGE/NV <path>`, `ME/NV <path>`	Like `MERGE`, but leaves the cursor at its original position after the insert.

Paths are resolved against the CMline working directory (see §6.11). Use absolute paths to be explicit. Files are read as UTF-8 with bad bytes replaced, so a binary or wrongly-encoded file will import as text-with-junk rather than failing the command.

The `DIR` dialog supports XyWrite-style **express keys**: type any printable characters and the cursor jumps to the first entry whose name contains the buffer (case-insensitive substring). Backspace shortens the buffer and re-jumps; backspacing all the way to empty snaps the cursor back to the top of the listing. The footer shows the current buffer (e.g. `42 entries in C:\Users\you - find: foo`) whenever it's non-empty. Esc with a non-empty buffer clears it; Esc with an empty buffer closes the dialog. `F3` finds again — it advances the cursor to the next row matching the current buffer, wrapping back to the top of the listing if there are no more matches below.

`Ctrl+Enter` on a highlighted file hands the path to the operating system's launcher: data files open in their associated app (a `.pdf` in your PDF viewer, a `.docx` in Word, etc.), and executables (`.exe`, `.bat`, `.com`, …) run directly. No confirmation prompt — XyWrite-style fire-and-forget. The dialog closes on a file launch; with a directory selected, the OS file manager opens at that path and the dialog stays put so you can keep browsing.

A bare **Ctrl tap** (press and release with no other key in between) copies the highlighted entry's fully-qualified path to the system clipboard, so you can paste it straight into another command, an editor pane, or another application. Either Ctrl key (left or right) works. Holding Ctrl while pressing any other

key — Ctrl+Enter, Ctrl+arrow, Ctrl+click — cancels the gesture, so modifier chords behave normally and never leak into a stray copy on release. The `..` parent row resolves to the parent directory's own absolute path. The dialog stays open after a copy so you can keep browsing or copy another path. The status bar echoes the copied path so you can confirm what landed on the clipboard.

`Del` on a highlighted file deletes it after a confirmation prompt. The delete is permanent (the file is unlinked, not sent to the recycle bin), matching the QPL `DEL` command's behavior. Directories are refused outright, and a file currently open in a tab is refused too — close the tab first (`AB` or `ABD`) and re-try. After a successful delete the listing re-scans in place, the entry count drops, and the cursor lands on whatever row took the deleted row's index.

7.27 Date, time, and stopwatch

Date, time, and elapsed-time helpers — handy for log entries, timing your work, and "what was I doing at" annotations.

Verb	Effect
`DA`	Insert the current date and time at the cursor, in the format last selected from *Tools* → *Insert date/time*.
`ZT`	Reset the elapsed-time stopwatch to zero.
`ET`	Insert the elapsed-time string at the cursor.
`ETS`	Show the elapsed-time string on the status line.

The stopwatch is a single global counter that starts when QuickNotes launches; `ZT` resets it. `$ET` reads the same value as a script variable (in seconds).

7.28 Preview and PDF export

Toggles for the on-screen display and commands for exporting the active note to PDF.

Verb / Form	Effect
`LN`	Toggle the line-number gutter in the editor.
`PRINTS`, `RV`, `TYS`	Toggle the Markdown preview pane. (The command `RV` is distinct from the special variable `$RV`.)
`PDF [path]`	Export the active note as a plain-text (monospaced) PDF. Synchronous.
`PDFS [path]`, `PDFSTYLED [path]`	Export the active note as a styled PDF (Markdown-rendered, using the Styled-PDF font). Runs asynchronously; chain with `WAIT` if a follow-up command needs the file on disk.

`PDF` always uses the bundled monospace font (Courier). `PDFS` / `PDFSTYLED` use the Styled-PDF font selected under *Options* → *Styled-PDF* → *Font…* and honor the margin settings described in §11.5.

The margin offset command `OF` is documented with the rest of the UI-sizing knobs in §7.20.

7.29 Sequencing

Commands for pausing script execution and waiting for keystrokes or asynchronous work to complete.

Form	Effect
`WAIT`	Block until the most recent asynchronous command (e.g. `PDFS`) has completed.
`WAIT <ms>`	Pause execution for `<ms>` milliseconds (1/1000 sec).
`READKEY <var> [prompt]`	Pause for one keystroke; store its character / keysym in `$<var>` and its scan code in `$SC`. The optional `[prompt]` is shown on the status line while waiting. Use `\n` or `\r` to create multi-line prompts.

Form	Effect
RK <var> [prompt]	Alias for READKEY.
READKEY/NV <var>, RK/NV <var>	Silent variant: no dialog, no prompt. Waits invisibly for one keystroke and stores it in $<var> / $SC. Pair it with a preceding PR or MSG when you want a visual cue under your own control.

READKEY / RK consume exactly one keystroke; chord modifiers (Shift, Ctrl, Alt) come back as separate events and don't count. The captured value in $<var> is the single character for printable keys (A, 0,) or a Tk keysym name for function and editing keys (F5, Return, Escape, BackSpace, Right, ...). $SC carries the underlying platform scan code, which scripts can use to distinguish keys that map to the same character (e.g. the main-row 1 vs the numpad 1).

All four forms (READKEY, RK, READKEY/NV, RK/NV) are transparent to focus: whatever widget had keyboard focus before the read — editor, CMline, notes panel — gets it back after the keystroke is captured.

7.30 Scripting (see chapter 8)

The following verbs make up the QPL scripting language proper. Chapter 8 covers each in detail.

Verb	Role
SET	Assign a variable
SV	Save selection to a named variable
LISTVAR	List script variables currently in memory
LOCAL	Declare a sub-scoped variable
IF / IFNOT	Conditional execution
ELSE / ELSEIF / ENDIF	Conditional block continuations
WHILE / ENDWHILE	Loop
BREAK / CONTINUE	Exit / restart the innermost enclosing WHILE

Verb	Role
GOTO	Jump to a :label
SUB / ENDSUB	Define a subroutine
CALL	Invoke a subroutine
RETURN	Return from a subroutine (with optional value into $RV)
QUIET	Suppress non-error status messages
SETERR / CLERR	Set / clear the trappable $ERROR flag
RUN	Run a .qns script file by name
RUNHERE	Run the fenced code block under the cursor

7.31 User libraries (see chapter 9)

Verb	Role
LOADLIB	Load (or reload) the user library.
ADDLIB	Additively merge a .qnu into the current library.
RELOADLIB	Replay the full LOADLIB + ADDLIB sequence.
UNLOADLIB	Drop every user-library routine.
LISTLIB	List loaded routines with their DOC strings.
EDITLIB	Open the user.qnu file in a new tab.

7.32 Key bindings (see chapter 10)

Verb	Role
LOADKEY	Bind a command to an available key (press-a-key dialog).
UNLOADKEY	Remove a binding.
LISTKEYS	List current bindings.
SAVEKBD	Save current bindings to a .qnk file.
LOADKBD	Merge bindings from a .qnk file.

Chapter 8

QPL: The Scripting Language

QPL — the QuickNotes Programming Language — is the language you write your scripts in. Every CMline verb is part of QPL, but the verbs often used in scripts are the ones that *don't* drive the editor directly: variables, expressions, control flow, subroutines. This chapter covers them all.

The chapter is organized in three parts. The first (this one, the longest) covers *data*: how to store values in variables, how to compute new values from old ones, and how to work with strings. Sections later in the chapter cover control flow (`IF`, `WHILE`, `GOTO`) and subroutines, plus the error model and the `QUIET` directive.

8.1 Variables

A variable in QPL is a named slot that holds a string or numerical value. You set one with `SET`, and read it back with `$NAME`. Variable names are case-insensitive: `$name`, `$Name`, and `$NAME` all refer to the same slot.

There are two forms of `SET`:

```
SET name = expression
SET name   literal text without a leading `=`
```

The first form (with =) evaluates the right-hand side as an *expression* (see below) and stores the result. The second form stores the rest of the line verbatim, with no expression parsing:

```
SET who = "world"
SET greeting = "Hello, " . $who
SET reminder Don't forget the milk
```

After those three lines, `$who` is `world`, `$greeting` is `Hello, world`, and `$reminder` is `Don't forget the milk`.

Variable names follow C-like rules: a letter or underscore followed by letters, digits, or underscores. A handful of names are reserved — these *special variables* are extremely useful in scripts (see below and Appendix B). `SET` refuses to write to them.

8.1.1 SV — save selection to a variable

A common shortcut for capturing reusable text: select a span in the editor, then run `SV <name>` to save its contents in `$<name>`. The verb (and the underlying idea) come from XyWrite. `SV foo` is equivalent to `SET foo = SEL()` but quicker to type and easier to remember when you're working at the CMline rather than in a script.

```
[save a selected snippet of text, then cancel the selection
SV sig; XD
[later, anywhere:]
WR $sig        ; pastes the snippet at the cursor
```

Rules: exactly one argument (the name); leading `$` is cosmetic (both `SV foo` and `SV $foo` work); names matching a read-only special variable are rejected; an empty selection trips `$ERROR` and leaves the variable untouched.

8.1.2 `LISTVAR` — what's in memory

`LISTVAR` pops a TXT dialog showing every script variable currently set, alphabetical by name (with positional args `$1`, `$2`, … sorted numerically first), each on its own line with its current value. Long values are truncated for display at 80 characters; newlines and tabs are shown as `\n` / `\t`. Pure info command — doesn't disturb `$ERROR`.

`LISTVAR` and `SV` together turn QuickNotes into a serviceable snippet manager: collect text bits with `SV`, browse them with `LISTVAR`, paste them with `WR $<name>`. The variables live for the rest of the session; if you want them to survive restart, `WRITEFILE` the relevant variable to a `.txt` and re-`READFILE` it next time, or stash the value inside a `SUB ... ENDSUB` block in your `user.qnu` library.

8.2 $-substitution

Anywhere a `$NAME` appears in a command — argument, expression, substring — it's replaced with the variable's current value at dispatch time:

```
SET who = world
PR Hello, $who!
```

prints `Hello, world!` to the status line.

Two escape forms are honored during substitution:

- `\$` is a literal `$`. Use it when you want a dollar sign that isn't a variable reference: `PR Price: \$5.99`.
- `\\` inside a `"..."` string literal is a literal backslash; the expression tokenizer handles this when it gets the value.

Substitution is recursive in name but not in value: a `$`-token inside the *value* of a variable does NOT get re-substituted. So if `$msg` holds the literal three characters `$x`, then `PR $msg` prints `$x`, not the value of `$x`.

Inside a `"...` string literal, embedded backslashes and double quotes in a substituted value are automatically escaped so the value can't break out of the surrounding quotes. This is why `STRIP("$SE")` is robust to selections that contain quotes.

8.2.1 When to quote `$var`

`$`-substitution is **textual** and happens **before the source line is parsed**. If `$var` holds `Hello`, the line `PR $var` literally becomes `PR Hello` and is then handed to the parser. Once you have that mental model, the three forms a value can appear in fall into place.

Bare `$var` — the default. Use it when:

- The command takes the rest of the line literally and any substituted text is safe: `PR $var`, `MSG $var`, `WR $var`, `WRL $var`, `TXT $var`. Everything after the verb is the payload — no tokenization, no quoting concerns.
- You're inside an expression and the value is numeric or a single unquoted word that the parser won't mistake for syntax: `SET total = $count + 1`, `IF $count == 0`, `WHILE $i < 10`.
- The value is known to be well-behaved — no embedded spaces, no quotes, no backslashes, no slashes that would collide with a separator. A single-word search pattern is fine: `SE /$keyword/`.

Quoted `"$var"` — turn the value into a string. Use it when:

- An expression context needs a *string*, not a word that could be interpreted as a scripting keyword or variable name. Without quotes, the expression parser tries to interpret the substituted text as syntax: `IF $name == Alex` fails because the parser sees `Alex` as a name to look up, not as the literal string `Alex`. `IF "$name" == "Alex"` does the right thing.
- A string-function argument: `SET n = LEN("$line")`.
- A `SET = "..."` assignment where you want a single string

value with embedded spaces or special characters: `SET path = "$drive/$folder"`.

Inside `"..."` strings, `$`-substitution still happens *and* QPL escape sequences (`\n`, `\t`, `\\`, `\"`, …) are activated. The quotes don't disable substitution — they just wrap the result as a single string token.

`VAR("name")` and `SEL()` — bypass the source-line lexer. Use these when the value might contain characters that would mis-parse if substituted textually:

- The value contains its own double quotes. If `$line` holds `He said "hi"`, then `PR "$line"` substitutes to `PR "He said "hi""`, which has unbalanced quotes. `STRIP(VAR("line"))` and `WR VAR("line")` hand the value directly to the evaluator, skipping the textual paste.
- The value contains control characters (tabs, newlines, NUL bytes).
- The current selection in a search pattern: `SE /$SE/` works only when the selection has no `/`. `SET pat = SEL()` followed by `SE /$pat/` is safer because `SET = SEL()` flows the value through the expression layer first.

Quick rules of thumb

"Printing or writing a value" → bare `$var` usually fine.
"Numeric / comparison expression" → bare: `IF $count > 0`.
"Comparing strings or building a string" → quote both sides: `IF "$os" == "windows"`.
"Value might have quotes / control chars / mismatched separators" → reach for `VAR("name")` or `SEL()`.
Diagnostic: if unsure, `MSG $var` or `PR $var` shows you exactly what gets substituted. If that text would re-parse cleanly downstream, bare works.

8.3 Arrays

QPL doesn't have a separate array type. Instead, numbered variables behave like an array — `$arr_0`, `$arr_1`, `$arr_2`, and so on — and a bracket-sugar form lets you index by number or by another `$`-variable:

```
$arr[3]
$arr[$i]
```

`$arr[3]` is shorthand for `$arr_3`. `$arr[$i]` evaluates `$i` first, then indexes the array with the result — so if `$i` is `3`, `$arr[$i]` reads `$arr_3` just like the literal form.

The `SPLIT` and `SPLITS` verbs (chapter 5) build arrays this way: `$name[0]` holds the count, `$name[1]` through `$name[N]` hold the elements.

You can also build a numbered array by hand:

```
SET fruits_0 = 3
SET fruits_1 = apple
SET fruits_2 = banana
SET fruits_3 = cherry
PR Got $fruits[0] fruits; first is $fruits[1].
```

8.3.1 Clearing an array

Because the storage layout is just a family of scalars (`$name_0`, `$name_1`, …), an array doesn't go away when you reassign the bare name. `SET arr = "hello"` overwrites `$arr` but leaves `$arr_0`, `$arr_1`, … exactly as they were. Two situations where this matters in practice:

- A routine runs a SPLIT into `arr`, the caller calls the routine again, the second call doesn't re-SPLIT (maybe it takes a different IF branch), and `$arr[1]` still reads back the first call's value.
- A top-level script runs `SPLIT arr "$x" ","`, then the user

runs another script later that expects a fresh `arr` and reads `$arr[1]` before assigning to it.

QPL gives you two ways to deal with this:

`CLEAR <name> [<name> …]` — explicit zero-out. For each name, this removes the scalar `$<name>` *and* every numbered slot (`$<name>_0`, `$<name>_1`, …). Names that have nothing to clear are not an error — `CLEAR` is the idempotent "make it empty" verb. Use it at the top of a routine or script that needs a guaranteed-empty starting state without going through `LOCAL`:

```
CLEAR arr tok line
SET arr = SPLIT "$txt" "\n"
```

`LOCAL <name>` — inside a CALL'd sub or a library routine, `LOCAL` is array-aware. It saves and clears every existing `$<name>_<digit>` slot in addition to the scalar `$<name>`, *and* sweeps out any *new* slots the routine creates inside the LOCAL'd scope (e.g. via a SPLIT) at exit time. So:

```
SUB process
    LOCAL arr
    SET arr = SPLIT "a,b,c" ","
   # $arr[1] is always "a" here, regardless of caller.
    PR Got $arr[1]
ENDSUB
```

— neither sees the caller's prior `$arr_*` nor leaks its own back. `LOCAL` is the right default inside routines; `CLEAR` is for the top-level / cross-routine cases where `LOCAL` doesn't apply.

A successful `SPLIT` or `SPLITS` *also* clears stale slots of the same name as a side effect (the sweep is pattern-based on `$<name>_<digits>`, so direct slot writes get caught too). What it *doesn't* do is help if your code path skips the SPLIT entirely — that's exactly where `CLEAR` and array-aware `LOCAL` come in.

8.4 Expressions

The `SET name = expr` form evaluates the right-hand side as an expression. So do the `IF` and `WHILE` keywords (sections 8.8, 8.9). QPL's expression evaluator handles numbers, strings, comparisons, logical operators, and function calls.

Numeric and string literals. Integer and decimal numbers are written the obvious way (`42`, `3.14`, `-5`). Strings are double-quoted: `"hello"`. Single quotes are not used.

Operators, lowest to highest precedence:

Group	Operators
Logical	`OR`, then `AND`, then `NOT`
Comparison	==, !=, <, >, <=, >= (numeric first, then string)
Additive	+, −, `.` (the `.` is string concatenation)
Multiplicative	`*`, `/`, `%`
Unary	unary −

So `2 + 3 * 4` is `14`, not `20`, and `"Hello, " . $who` is string concatenation. Comparisons return `1` for true and `0` for false:

```
SET n = 7
SET odd = $n % 2 == 1      # 1 (true)
SET twice = $n * 2         # 14
SET full = $n . " items"   # "7 items"
```

Comparisons try numeric first; if both sides parse as numbers, you get a numeric compare. Otherwise you get a string compare: `"10" < "9"` is true (string), `10 < 9` is false (numeric).

8.5 String literals and escape sequences

Inside a `"...."` string literal, the following escape sequences are honored:

Escape	Meaning
`\n`	newline
`\t`	tab
`\r`	carriage return
`\0`	NUL byte
`\\`	literal backslash
`\"`	literal double quote
`\xHH`	one byte, two hex digits
`\dNNN`	one byte, three decimal digits (0..255)
`\uHHHH`	Unicode code point, four hex digits (BMP)

Unknown escapes pass through verbatim: `\d` outside a `\dNNN` sequence is the literal two characters `\` and `d`. The same escape set is honored by the text-payload commands `PR`, `MSG`, `TXT`, `WR`, `WRL`, `INPUT` (prompt half), and by `SPLIT` / `SPLITS` arguments — see chapter 6 for the cross-command summary.

8.6 String functions

The expression evaluator includes a small library of string functions. They take their arguments in parentheses:

Function	Returns
`STRIP(s)`	`s` with leading and trailing whitespace removed
`LSTRIP(s)`	`s` with leading whitespace removed
`RSTRIP(s)`	`s` with trailing whitespace removed
`TRIMLEFT(s, n)`	`s` with the first `n` characters removed
`TRIMRIGHT(s, n)`	`s` with the last `n` characters removed
`UPPER(s)`	`s` in upper case
`LOWER(s)`	`s` in lower case
`LEN(s)`	number of characters in `s`
`SUBSTR(s, i, n)`	substring starting at offset `i` (0-based), length `n`

Function	Returns
`SEL()`	the current selection text (same value as `$SE`, but as a proper string value)
`VAR(name)`	the value of `$<name>`, as a proper string value

`SEL()` and `VAR()` are useful when a value might contain characters (like quotes or backslashes) that would mis-parse if substituted into the source line directly. `SEL()` returns the same thing as the `$SE` reserved variable but at evaluation time, so it never goes through the source-line lexer.

The full function table is in Appendix C.

8.7 Special variables, at a glance

QPL reserves a number of names for *special variables* — read-only values that reflect QuickNotes' current state. You met `$SE` and `$ERROR` earlier; here are the ones that come up most in scripts:

Variable	Value
`$SE`	current selection (4 KB cap)
`$ERROR`	`1` if the last error-aware command failed, else `0`
`$FI`	active note's filename (no path)
`$FP`	active note's full path
`$PA`	active note's directory
`$CWD`	CMline working directory
`$OS`	host OS: `Windows`, `Darwin`, or `Linux`
`$RC`	last shell command's exit code
`$RV`	sub/library routine return value
`$DA`, `$TM`	current date / time
`$PS`	persistent-selection state (1 / 0)
`$CF`, `$CFL`	active config file / last `LoadCfg` file
`$1`..`$#`	positional script arguments (see chapter 8C)
`$*`	all positional arguments joined

The full list is in Appendix B with an example of each.

Why the case insensitivity?

QPL inherits XyWrite's case-insensitive convention for both command verbs and variable names. `SET who`, `set Who`, and `set WHO` all write to the same slot, and `$who` reads it from any of them. The values held by variables, of course, preserve case exactly as you wrote them.

8.8 Branching with `IF`

`IF` makes the next action conditional on an expression. QPL has two forms — a single-line form for one quick conditional action, and a block form for several lines under one condition:

```
IF $n == 0 PR Nothing to do.
```

```
IF $n == 0
    PR Nothing to do.
    RETURN
ENDIF
```

In the single-line form, the rest of the line *after* the expression — with no intervening semicolon — is the action. The action runs if the expression evaluates as true and is skipped otherwise.

In the block form, every line between `IF` and `ENDIF` is the action. `ENDIF` (or its alias `EI`) closes the block. Indentation is for the human reader; QPL doesn't care about it.

Nested `IF...ENDIF` blocks are supported.

Semicolons turn `IF` into block form

A common mistake is to write `IF cond ; cmd` as a one-line shorthand. It doesn't work that way: the `;` is a command separator that splits the line into two dispatched commands. The first, `IF cond` (no trailing text), opens a block-form `IF` that requires a matching `ENDIF` later –

which the second half doesn't provide. The result is a "missing ENDIF" complaint when the script ends.
If you want everything on one line, either use the bare single-line form (`IF cond cmd`, no semicolons) or explicitly close the block: `IF cond ; cmd ; ENDIF`.
Also: `IF` relies on its left operand being a real value. If the variable in your condition is undefined, `$name` substitutes to an empty string and the expression becomes something like `== 0` – which fails to parse, with the same "missing ENDIF" surface symptom. Defend with a `SET` before the `IF`, or test with `IF "$name" == ""` (quoted, so the empty string is a valid string operand).

The full block form adds `ELSE` and `ELSEIF`:

```
IF $score >= 90
    PR Excellent.
ELSEIF $score >= 70
    PR Passing.
ELSE
    PR Try again.
ENDIF
```

`ELSEIF` and `ELSE IF` (two words) are accepted interchangeably. Each gets its own expression; the first branch whose expression is true runs, and the rest are skipped.

Truthiness. An expression result evaluates as true if it's a non-zero number, or a non-empty string that doesn't parse as zero. So `0`, `""`, and the literal string `"0"` are false; everything else — including `"false"` and `"no"` — evaluates as true. The error flag `$ERROR` is `1` or `0`, so `IF $ERROR` reads naturally.

`IFNOT` is shorthand for `IF NOT`:

```
IFNOT $ERROR PR Last command succeeded.
```

Both IF and IFNOT accept either form (single-line or block).

8.9 Looping with WHILE

WHILE runs a block of commands repeatedly while a condition stays true:

```
SET i = 1
WHILE $i <= 5
    PR Iteration $i.
    WAIT 1000
    SET i = $i + 1
ENDWHILE
```

ENDWHILE (or its alias EW) marks the bottom of the loop. The condition is re-evaluated at the top of each iteration; if it starts out false, the body is skipped entirely.

The loop body can include any QPL commands, including IF blocks, nested WHILEs, CALLs to subroutines (see 8C), and so on.

At the CMline (or anywhere else outside a running script), WHILE, ENDWHILE, and EW report a clear error rather than silently doing nothing. They are also harmless no-ops inside a skipped IF branch — the same convention as ENDSUB / ES — so a commented-out IF 0 ... ENDIF block containing a loop doesn't spam the status line.

8.9.1 BREAK and CONTINUE

BREAK exits the innermost enclosing WHILE, jumping to the line after its ENDWHILE. CONTINUE jumps back to the WHILE line so the condition re-evaluates, skipping the rest of the current iteration's body. Both work only inside a script (or a library routine) — used at the CMline they report an error.

Typical use: walk through every match of a pattern in the current buffer,

bailing out the moment `SE` reports no more matches. `SE` (without `/T`) starts each search from the current cursor position, which lands at the end of the previous match — so each iteration advances:

```
WHILE 1
    SE /TODO/
    IF $ERROR
        BREAK
    ENDIF
    PR Found one.
ENDWHILE
```

The equivalent without `BREAK` (using a sentinel variable, the old idiom) is more verbose:

```
SET keep_going = 1
WHILE $keep_going
    SE /TODO/
    IF $ERROR
        SET keep_going = 0
    ELSE
        PR Found one.
    ENDIF
ENDWHILE
```

Watch the `/T` modifier here: `SE/T  /TODO/` forces every iteration to restart at the top of the file, so on a file with any match the loop never advances — it just keeps re-finding the first occurrence. Use bare `SE` inside a walk loop.

`CONTINUE` is useful when most iterations want to skip the rest of the body once a condition is met — for example, “process only lines that start with a header marker”:

```
SET i = 0
WHILE $i < $count
```

```
    SET line = $lines[$i]
    SET i = $i + 1
    IF SUBSTR($line, 0, 1) != "#"
        CONTINUE
    ENDIF
    PR Header: $line
ENDWHILE
```

Note that `SET i = $i + 1` runs *before* the `IF`/`CONTINUE` check. Putting the increment after `CONTINUE` would skip it on every iteration that hits the continue, and the loop would never terminate.

Technical Note: Placement and the IF stack

BREAK and **CONTINUE** also roll back any block-form IF frames that were opened inside the loop body, so the post-loop or re-iterated code starts with a clean stack. Two related patterns work cleanly:

- *Bare* (`BREAK` on its own line, or at the top level of the loop body).
- *Single-line IF* (`IF cond BREAK`): the single-line form of `IF` doesn't push a block frame, so there's nothing to roll back.
- *Block-form multi-line IF*: `IF cond` / next line `BREAK` / next line `ENDIF`. The `BREAK` rolls back the `IF` frame on its way out; the `ENDIF` on the following line is jumped over by the new PC.

One pattern does *not* work: chaining an entire block-form `IF` onto a single line, like `IF cond ; BREAK ; ENDIF`. The `BREAK` pops the `IF` frame, then the trailing `ENDIF` on the same line tries to pop again and reports "ENDIF without matching IF." Use `IF cond BREAK` or a multi-line block instead.

8.10 GOTO and labels

A label is a line starting with `:` followed by an identifier:

```
:start
```

`GOTO :name` jumps execution to the named label. Labels are case-insensitive and have script-global scope: a `GOTO` inside one `IF` block can target a label in another block, or before / after the block entirely.

```
SET n = 0
:loop
    SE /TODO/
    IF $ERROR
        GOTO :done
    ENDIF
    SET n = $n + 1
    GOTO :loop
:done
MSG Found $n TODOs.
```

`GOTO` is the simplest tool for the "loop until something stops being true" pattern, and it's the standard way to exit early from a deeper construct. Outside those cases, prefer `WHILE` or a subroutine — labels in many places make a script harder to follow.

Jumping to a label that doesn't exist is a fatal error (`_QplFatal`), not a trappable one — the typo gets caught immediately rather than silently doing nothing.

8.10.1 `GOTO` inside `IF` blocks

`GOTO` out of an `IF` body works correctly — the runtime pops any open `IF` frames between the `GOTO` and its target so that a "clean" GOTO out of a conditional doesn't leak a spurious "unclosed IF block" warning at script end. This relies on a static pre-scan of label depths, which treats each *physical* line independently. Two practical consequences:

- Put `IF`, the `GOTO`, and `ENDIF` on **separate lines** when the `GOTO`

lives inside a block-form `IF`:

```
IF $done
    GOTO :end
ENDIF
```

- Don't chain them on one line with semicolons (`IF $done ; GOTO :end ; ENDIF`). The pre-scan can't see inside semicolon-chained commands and the depth tracking will misjudge the nesting, leading to spurious warnings or mis-popped frames. The same caveat already applies to `BREAK` and `CONTINUE` inside `WHILE` loops.

The single-line form `IF cond GOTO :label` is fine — it doesn't open a block at all, so there's nothing to mis-track.

8.11 Line continuation

A `\` as the last non-whitespace character on a script line splices the line onto the next, so long commands don't have to fit on one physical line:

```
SET msg = "First half " \
        . "second half."
PR $msg
```

After splicing, the dispatcher sees a single `SET msg = "First half " . "second half."`. The `\` and the line break are consumed; the next line's leading whitespace is preserved verbatim, which lets you indent the continuation to taste.

Continuation is suppressed inside a `"..."` string literal — embed a real newline with `\n` or break the string and concatenate with `.`. The CMline itself is always a single line, so continuation is a script-source feature only.

8.12 Scripts on disk

A `.qns` file is just a sequence of CMline commands, one per line, saved into the `scripts/` folder next to `quicknotes.py`. The *Scripts* menu auto-populates from that folder; clicking a `.qns` file runs it. Equivalently, from the CMline:

```
RUN <name>
```

`<name>` is the script's filename without the `.qns` extension. Paths work too: `RUN ../shared/stamp` reaches outside the scripts folder.

Arguments after `RUN` become `$1`, `$2`, …, `$#` (the argument count) and `$*` (all arguments joined). So `RUN greet world friend` runs `greet.qns` with `$1 = world`, `$2 = friend`, `$# = 2`, and `$* = greet world friend`.

For ad-hoc scripts that don't deserve their own file, `RUNHERE` (bound to **Alt+Enter** in the editor) runs the fenced code block your cursor is currently inside. The fence's language tag must be `qns`, `qpl`, or `quicknotes`. So a block embedded in a note looks like this:

````
```qpl
SET who = world
MSG Hello, $who!
```
````

Placing the cursor anywhere between the opening fence and the closing fence, then pressing **Alt+Enter**, runs the block as a script. `RUNHERE` from the CMline does the same thing:

```
RUNHERE
```

The cursor needs to be inside a `qns`-fenced block in the editor for `RUNHERE` to do anything; otherwise `Alt+Enter` falls back to "run the current line silently *as a shell command*" (its everyday meaning when no fenced block is in scope).

8.13 startup.qns

If a file named `startup.qns` exists in the directory with `quicknotes.py` when QuickNotes launches, it's run once after the user library auto-loads and before the first interactive command. This is the place for anything you'd want set up every session: variables, `LOADKBD` to restore a saved key-binding profile, `ADDLIB` to merge an extra library, a welcome `PR` or `MSG`.

A minimal startup.qns that loads a saved key-binding profile and greets you:

```
LOADKBD work.qnk
PR Welcome back.
```

`startup.qns` is just an ordinary script; if it triggers a non-fatal error, the error is reported on the status line but startup continues.

8.14 Subroutines

A subroutine is a named block of QPL code that can be invoked from anywhere else in the same script. Define one with `SUB` and `ENDSUB`:

```
SUB greet
    PR Hello!
ENDSUB

CALL greet
```

The `SUB greet` line names the routine; everything up to the matching `ENDSUB` is its body. `CALL greet` runs the body, then resumes at the line after the `CALL`. `ENDSUB` has the alias `ES`. A `SUB` name follows the same identifier rules as a variable name.

8.14.1 Arguments

A `CALL` can pass positional arguments separated by whitespace. Inside the sub, the arguments appear as the special variables `$1`, `$2`, …, up to `$#` — which holds the count. `$*` is all of them joined with single spaces between.

```
SUB greet
    IF $# == 0
        PR Hello, world!
    ELSE
        PR Hello, $1!
    ENDIF
ENDSUB

CALL greet
CALL greet Alice
```

Arguments are *positional*, not named, and they're all strings from the sub's perspective. There's no `$0` (the routine name itself isn't passed in).

Arguments containing spaces need to be quoted in the `CALL`:

```
CALL greet "Alice and Bob"
```

so `$1` becomes the three-word string `Alice and Bob` rather than `$1 = Alice`, `$2 = and`, `$3 = Bob`.

8.14.2 Returning a value

`RETURN [expression]` exits a sub. With an expression, the value is stored in the special variable `$RV` (Return Value), which the caller can read after the `CALL`:

```
SUB double
    RETURN $1 * 2
ENDSUB
```

```
CALL double 21
PR The answer is $RV.
```

`RETURN` with no expression leaves `$RV` unchanged from whatever it was before — useful for early-exit subs that don't have a value to communicate:

```
SUB process
    IF $# == 0
        PR process: nothing to do.
        RETURN
    ENDIF
    PR processing $1...
ENDSUB
```

Falling off the bottom of a sub (reaching `ENDSUB` without an explicit `RETURN`) is equivalent to `RETURN` with no expression.

8.14.3 Early-exit shorthand: `EXIT` / `EX`

`EXIT` (alias `EX`) is the value-less early-out:

- **Inside a CALL'd sub**, `EXIT` pops the sub frame and resumes the caller right after the `CALL` — equivalent to `RETURN` with no expression. `$RV` is left as whatever it was before.
- **At script top level** (no enclosing `CALL` frame), `EXIT` ends the running script cleanly — `RUN myscript` returns at that point, even if there's more text below the `EXIT` line.

So `EXIT` is "stop running this thing now," regardless of whether "this thing" is a sub or the top-level script. Useful for early sanity checks:

```
IF $# == 0
    PR Need at least one argument.
    EXIT
ENDIF
```

That works at script top level without having to wrap the rest of the script in an `IF` block.

8.14.4 Exit out of everything: `EXITALL` / `EX1`

`EXITALL` (alias `EX1`) goes further than `EXIT`: it unwinds *every* running script all the way up to the CMline. From inside a sub that's three levels deep in `CALL`s, four levels deep in nested `RUN`s, doesn't matter — `EXITALL` ends all of them at once.

Useful when a deeply-nested condition makes everything above it moot: instead of threading matching `EXIT` (or `RETURN`) calls back up the chain, a single `EXITALL` cleans the slate.

```
SUB validate
    IF $1 == ""
        PR Required argument missing.
        EXITALL
    ENDIF
ENDSUB
```

Here, a missing argument in `validate` aborts not just the sub but the entire script that called it (and any script that called *that*, and so on). Each affected script reports "finished" normally on the status line — `EXITALL` is a clean exit, not an error.

`EXITALL` does **not** close QuickNotes — use `QUIT` for that.

`EXIT` vs. `EXITALL` vs. `QUIT`

Three escape hatches with different reach:

- `EXIT` / `EX`: exit the *current* sub (or end the current script if at top level).
- `EXITALL` / `EX1`: end *every* running script – nested CALLs and RUNs alike – back to the CMline.

- `QUIT`: close the QuickNotes window (prompts per modified tab). `QUIT/NV` skips the prompts and discards unsaved changes silently.

8.14.5 Scope: `LOCAL`

By default, all variables in QPL are *script-global*: a `SET x = 5` inside a sub writes to the same `$x` the caller sees, which means a careless sub can clobber its caller's data.

`LOCAL` declares one or more variables as local to the current sub invocation. A LOCAL variable shadows any same-named global for the duration of the call, and is automatically restored when the sub returns. The syntax accepts **one or more space-separated names** on a single line:

```
LOCAL x                  ; one variable
LOCAL i j tmp          ; multiple variables in one go
LOCAL count   note   ; whitespace between names is fine
```

It's idiomatic to declare every variable you intend to use as scratch in a sub on a single `LOCAL` line at the very top of the sub body. That keeps the sub's namespace contract obvious at a glance.

A worked example:

```
SET x = 100

SUB swap_and_increment
    LOCAL x tmp
    SET x = 5
    SET tmp = "scratch"
    PR Inside the sub, x is $x, tmp is $tmp.
ENDSUB

CALL swap_and_increment
```

```
PR After the sub, x is $x.
```

This prints:

```
Inside the sub, x is 5, tmp is scratch.
After the sub, x is 100.
```

The sub gets its own private `$x` and `$tmp`. The global `$x` is unchanged; the global `$tmp` (if any) is unchanged; a `$tmp` that didn't exist before is removed entirely on return rather than left as `""`.

`LOCAL` works in two places:

- Inside a `SUB ... ENDSUB` block invoked with `CALL` (the classic case).
- At the top level of a library routine, i.e. inside the body of a `.qnu` `SUB ... ENDSUB`. Library routines run as flat scripts but they're conceptually sub-like, and `LOCAL` is restored on routine exit the same way it is on `RETURN` / `ENDSUB` in a regular sub. This is what keeps a `.qnu` library's scratch variables from cluttering the caller's namespace.

Outside both of those contexts (the bare CMline, the top level of a `.qns` script), `LOCAL` is a no-op with a status-line warning.

Mental model

Think of `LOCAL x y z` as "save the current values of `$x`, `$y`, and `$z`; let me have fresh slots until the routine exits; then put the saved values back." That's literally how it's implemented.

`LOCAL` is array-aware. When you `LOCAL <name>`, every existing `$<name>_<digit>` slot is also saved and cleared — not just the scalar `$<name>` — and any *new* numbered slots created inside the LOCAL'd scope (for example, by a `SPLIT` into the LOCAL'd name) are swept out when the routine exits. So a routine that does

```
SUB process
```

```
    LOCAL arr
    SET arr = SPLIT "a,b,c" ","
    PR Got $arr[1]
ENDSUB
```

neither sees the caller's prior `$arr_*` nor leaves its own behind. This matters whenever a routine that takes a different code path on different calls (e.g. some calls SPLIT, others don't) might otherwise see stale slots from a previous call. Without the array-aware behaviour, `LOCAL arr` would null the scalar `$arr` but leave `$arr_0..$arr_N` untouched, so `$arr[1]` in the second call would still return the first call's data.

For places where `LOCAL` doesn't apply — top-level `.qns` scripts, the bare CMline, or a deliberate cross-routine reset — see the `CLEAR` verb in chapter 8.0, "Arrays".

8.15 The error model

Commands in QPL come in two flavours when something goes wrong.

Trappable failures are conditions that the caller might reasonably want to recover from: a search not finding its pattern, a `READFILE` of a missing file, an `INPUT` cancel. These set the special variable `$ERROR` to `1` and leave the script running. The script can branch on `$ERROR` to decide what to do next.

Fatal failures are bug-grade errors that indicate the script itself is broken: an unknown command name, a `GOTO` to a label that doesn't exist, a division by zero in an expression, a `CALL` to a sub that doesn't exist. These raise an internal `_QplFatal` exception that aborts the script immediately and pops a dialog with the error message. There's no way to trap a fatal inside QPL itself — the design assumes you want to find and fix the bug rather than paper over it.

The `$ERROR` flag is automatically cleared at the start of every action-

command dispatch, so it always reflects the last error-aware command and nothing earlier:

```
SE /foo/
IF $ERROR
    PR Foo not found.
ENDIF
```

A few verbs — control-flow keywords (`IF`, `WHILE`, `GOTO`, `SET`, etc.) and the housekeeping ones (`QUIET`, `DX`, `DO`) — deliberately *don't* touch `$ERROR`. They're meant to test or manage state, so they leave the flag alone.

8.15.1 `SETERR` and `CLERR`

`SETERR` and `CLERR` set and clear the `$ERROR` flag manually. You'll reach for them in scripts that compute their own success / failure conditions:

```
SUB find_marker
    SE /[[MARKER]]/
    IF $ERROR
        RETURN
    ENDIF
    IF $LN > 1000
     PR Marker found but too far in; treating as failure.
        SETERR
        RETURN
    ENDIF
ENDSUB
```

Both are in the no-touch list, so calling either one doesn't itself clobber the flag you just set or cleared.

8.16 QUIET

By default, most CMline commands write a brief status line about what they did: `SE: found`, `CH: replaced 12 occurrences`, `SA: saved`. These are helpful when you're typing commands by hand, but they can clutter the status line in a script that runs many small operations.

`QUIET` suppresses non-error status messages from that point in the script until the script ends. Errors still report normally, so a `QUIET` script is silent on success and loud only when something goes wrong:

```
QUIET
SE /TODO/
IF $ERROR
    PR No TODOs in this file.
    RETURN
ENDIF
```

`QUIET` is scope-bound to the current script: it doesn't leak into a sub-script invoked via `RUN`, and it's automatically cleared when the outer script ends. `QUIET` and its stronger sibling `QUIET/NV` (below) both accept an optional `ON` or `OFF` argument so a script can disable them mid-run when verbose output is wanted again:

```
QUIET                          ; or: QUIET ON
... silent block ...
QUIET OFF
... verbose block ...
```

You can always `PR` your own progress messages — those are explicit and not suppressed by `QUIET`.

8.16.1 QUIET/NV — total silence

`QUIET/NV` is the stronger form: it suppresses **all** editor chatter, including the non-fatal error messages that plain `QUIET` lets through. The only editor-

emitted output that still appears is a fatal, script-halting error — those pop a modal abort dialog and always survive. `QUIET/NV` is the right choice when a script wants to handle every recoverable failure itself via `$ERROR` and present no other UI to the user.

`PR` is exempt: it's the script author's deliberate output verb, not editor chatter, so a `PR Press a key...` prompt still displays even with `QUIET/NV` in force. Likewise `MSG`, `TXT`, and `INPUT` pop their own dialogs and are unaffected by either quiet flag.

Syntax mirrors `QUIET`:

```
QUIET/NV                        ; or: QUIET/NV ON
... totally silent block ...
QUIET/NV OFF
```

`QUIET` and `QUIET/NV` are independent flags: turning one off does not affect the other. Both reset to their pre-script values when the running script exits.

8.17 Putting it all together

Here's a script that uses every piece this chapter has covered. It walks the document, counts TODOs, and reports the result:

```
QUIET

SUB count_todos
    LOCAL n
    SET n = 0
    SE/T /TODO/
    WHILE NOT $ERROR
        SET n = $n + 1
        SE /TODO/
    ENDWHILE
```

```
    RETURN $n
ENDSUB

CALL count_todos
IF $RV == 0
    MSG No TODOs in this file.
ELSE
    MSG Found $RV TODOs.
ENDIF
```

The sub uses `LOCAL` to keep its counter private, leans on `$ERROR` to know when to stop walking, returns the count, and the caller branches on the result. `QUIET` at the top keeps the search noise off the status line while the loop runs.

That's the QPL chapter. Chapter 9 picks up the next layer: how to take routines you write and put them in a `.qnu` library file that QuickNotes auto-loads at startup, so your personal toolkit is available from any script you write.

Chapter 9

User Libraries

The QPL routines you find yourself reaching for again and again deserve a home outside the script files that use them. The *user library* is that home: a single text file (or several) full of named routines that QuickNotes auto-loads at startup, making each one available as a CMline verb. After a few weeks of using QuickNotes you'll likely have a small personal toolkit there — a timestamp-inserter, a word-count helper, a "wrap this selection in quotes" routine — that you reach for as if it were built in.

This chapter covers the `.qnu` file format itself, the verbs that manage libraries through their lifecycle, the bundled *system library*, and the rules for what happens when names collide.

9.1 The .qnu file format

A user library lives in a file with a `.qnu` extension. It's plain text — a sequence of `SUB`…`ENDSUB` blocks. The canonical example, drawn from QuickNotes' bundled system library:

```
SUB hello
   # DOC: Print a friendly greeting to the status line.
     PR Hello, world!
```

```
ENDSUB

SUB timestamp
   # DOC: Insert the current date and time at the cursor.
    WR $DA $TM
ENDSUB
```

The rules:

- Every routine is bracketed by `SUB <name>` and `ENDSUB` (or alias `ES`).
- `<name>` follows the same identifier rules as a variable.
- A line beginning `# DOC:` *inside* a routine body sets the routine's documentation string, which `LISTLIB` displays. Multiple `# DOC:` lines concatenate with newlines, so a long description can span several lines.
- Other `#` lines inside a routine are ordinary QPL comments and pass through to the runner unchanged.

`.qnu` files may also begin with a YAML-style frontmatter block (the same `---` … `---` header that QuickNotes' note files use). The frontmatter is tolerated and ignored, so you can save a library file as if it were any other note without confusing the parser.

9.2 Invoking a library routine

Once loaded (see *Loading and reloading* below), a library routine is callable from the CMline just like any built-in verb:

```
HELLO
TIMESTAMP
```

Arguments after the routine name become `$1`, `$2`, …, `$#`, and `$*` — exactly the same convention as `CALL` (chapter 8C). A routine's `RETURN` value lands in `$RV` for the caller to read:

```
SUB double
    # DOC: Double a number passed as an argument.
     RETURN $1 * 2
ENDSUB
```

After `DOUBLE 21`, `$RV` is `42`.

Routine names are case-insensitive, just like built-in verbs: `HELLO`, `hello`, and `Hello` all run the same routine.

9.3 Loading and reloading

QuickNotes looks for a user library at `scripts/user.qnu` on startup and loads it automatically. From then on, the lifecycle commands let you load, merge, drop, and reload libraries on the fly without restarting:

Form	Description
`LOADLIB [path]`	Load (or reload) the library. With no path, reads `scripts/user.qnu`. RESETS the load sequence: any prior `ADDLIB`s are forgotten.
`ADDLIB <path>`	Merge a second `.qnu` file into the existing library. Routine-name collisions are last-wins.
`RELOADLIB`	Re-parse and replay the full `LOADLIB` + `ADDLIB` sequence in order.
`UNLOADLIB`	Drop every user-library routine. System library unaffected.
`LISTLIB`	Pop a TXT dialog listing every loaded routine with its DOC string.
`EDITLIB`	Open `scripts/user.qnu` in a new tab for editing.

The typical write-and-test cycle:

1. `EDITLIB` to open the library in a tab.
2. Edit; add or change a `SUB`.
3. Ctrl+S to save.
4. `RELOADLIB` to pick up the changes.
5. Try the new routine from the CMline.

LOADLIB reports trappable failures via $ERROR: missing file, parse errors, a file that produced zero accepted routines (everything shadowed — see below). On parse error the previously loaded library is preserved, so a syntax slip during editing doesn't strand you with no library.

9.4 Routine-name shadowing

When LOADLIB or ADDLIB reads a .qnu file, every SUB name is validated against three other namespaces:

1. **Built-in CMline verbs.** A SUB SE would conflict with the built-in SE search verb — the loader rejects it and warns. Built-ins always win.
2. **Special variables.** A SUB ERROR would conflict with the special variable $ERROR. Likewise rejected.
3. **Other routines already loaded.** When ADDLIB brings in a routine whose name is already in the library, the new one replaces the old — *last-wins*.

Rejected routines from rules 1 and 2 are reported in the load status; the rest of the file loads normally.

9.5 The bundled system library

QuickNotes ships with a small *system library* compiled into quicknotes.py itself — a handful of routines (HELLO, TIMESTAMP, WHEREAMI) that demonstrate the format and provide useful starting examples. The system library is loaded at startup and lives alongside your user library; routine names in user libraries take priority over same-named system routines (so you can override any system routine with your own version without editing quicknotes.py).

UNLOADLIB drops user-library routines only — the system library

stays loaded. The system library can be disabled entirely by setting `system_library: false` in the config file, but that's rarely useful.

Tagged listing

LISTLIB tags each routine with its source: `[USER]` for routines from a user `.qnu` file, `[SYSTEM]` for the bundled ones. After several ADDLIBs the source path is also shown, so you can tell which file a given routine came from.

9.6 A worked example

Suppose you frequently want to wrap a selection in double quotes. Doing that by hand each time is tedious, so tuck it away in a routine. Open the library with `EDITLIB` and add:

```
SUB quote
   # DOC: Wrap the current selection in double quotes.
    IF $SE == ""
        PR quote: nothing selected.
        SETERR
        RETURN
    ENDIF
    LOCAL wrapped
    SET wrapped = "\"" . SEL() . "\""
    DN            # Delete selection ...
    WR $wrapped   # then replace it
ENDSUB
```

Save the tab, run `RELOADLIB`, and now `QUOTE` is a verb you can call from the CMline, chain in another script, or bind to a key (chapter 10).

A library file with thirty or forty such routines is the difference between QuickNotes-as-an-editor and QuickNotes-as-your-personal-toolbox.

Chapter 10

Key Bindings

Once you've built up a small toolkit of routines in your user library, the natural next step is to bind the ones you reach for most to keys. QuickNotes' key-binding facility lets you bind any single CMline command — or any `;`-chained sequence of them — to an available keystroke. The binding works app-wide regardless of which widget has focus, and it's persisted across sessions so the key you set today still does the right thing tomorrow.

10.1 Binding a key

The `LOADKEY` verb starts a binding:

```
LOADKEY <command>
```

`<command>` is anything you'd type at the CMline — a single verb, a verb with arguments, or a `;`-chained sequence. After Enter, a small dialog pops up:

> *Press the key combination to bind to this command:*
>
> `<command>`
>
> *Esc to cancel.*

Whatever key you press becomes the binding. If the key is already bound to another command, you'll get a confirm prompt; click *Yes* to replace, *No* to leave the old binding alone.

The command text is stored *verbatim*. Any `$NAME` in it is substituted at *key-press* time, not at `LOADKEY` time, so a binding to `PR Hello $name` always uses the current value of `$name` when the key fires, not whatever it was when you set the binding.

`LOADKEY` consumes the *rest of the source line* as its macro — so a `;`-chained sequence binds as-is, without `\;` escapes:

```
LOADKEY DX; SE /TODO/; PR Found; DO
```

That whole four-step chain becomes one keystroke.

10.2 What can you bind?

QuickNotes maintains two sets of keystrokes:

Reserved keys are the ones QuickNotes itself uses: `F1`, `F3`, `F5`, `F8`, `F9`, `F10`, `Esc`, `Alt+Enter`, `Ctrl+R`, `Ctrl+Shift+R`, and a few others. These can't be re-bound; the press-a-key dialog rejects them with a note explaining why.

Candidate keys — the ones you *can* bind — fall into four groups:

- Free function keys: `F2`, `F4`, `F6`, `F7`, `F11`, `F12`.
- Shift + any function key: `Shift+F1` … `Shift+F12`.
- Ctrl + Shift + any letter: `Ctrl+Shift+A` … `Ctrl+Shift+Z`.
- Alt + most letters: `Alt+B`, `Alt+C`, `Alt+D`, `Alt+G`, `Alt+I`, `Alt+J`, `Alt+K`, `Alt+L`, `Alt+M`, `Alt+N`, `Alt+P`, `Alt+Q`, `Alt+R`, `Alt+S`, `Alt+T`, `Alt+U`, `Alt+X`, `Alt+Y`, `Alt+Z`. The letters `A`, `E`, `F`, `H`, `O`, `V`, `W` are skipped because they're already in use by the menu bar (`Alt+F` opens the File menu, etc.).

That's about 75 keystrokes total — plenty of room for a personal toolkit.

10.3 Listing what you've bound

`LISTKEYS` pops a TXT dialog showing every current binding, sorted alphabetically by key:

```
User key bindings: 3

  <Alt-b>              QUOTE
  <Ctrl-Shift-T>       TIMESTAMP
  <F2>                 HELLO
```

The exact form of each key (with angle brackets and Tk modifier names) matches what `SAVEKBD` writes to a `.qnk` file and what `UNLOADKEY` accepts as an explicit argument.

10.4 Removing a binding

`UNLOADKEY` has three forms:

Form	Effect
`UNLOADKEY`	Pop a press-a-key dialog limited to currently-bound keys; pressing one removes its binding.
`UNLOADKEY <spec>`	Remove the named binding directly, e.g. `UNLOADKEY <F2>`.
`UNLOADKEY ALL`	Drop every user binding at once.

`UNLOADKEY ALL` is the safety net: if you accidentally bind something that steals input you can't otherwise reach, enter `UNLOADKEY ALL` on the CMline to start over.

10.5 Keyboard-profile files (.qnk)

A `.qnk` file is a plain-text snapshot of your bindings. The format is one binding per line:

```
<spec>  <command>
```

`#` starts a line comment; blank lines are ignored. A typical file:

```
# Daily work bindings
<F2>            HELLO
<Alt-b>         QUOTE
<Ctrl-Shift-T>  TIMESTAMP

# Note-cleanup macros
<Shift-F1>      DX; CH/T /\t/    /; DO
```

Two verbs manage `.qnk` files:

Form	Effect
`SAVEKBD <path>`	Write the current bindings to `<path>`. Adds `.qnk` if no extension.
`LOADKBD <path>`	Merge bindings from `<path>` into the current set (last-wins). To replace entirely, run `UNLOADKEY ALL` first.

`SAVEKBD` overwrites without prompting. `LOADKBD` reports any warnings (unknown spec, reserved spec, missing command text) in a TXT dialog so you can fix the file.

This is how profile-switching works: maintain a couple of `.qnk` files (`work.qnk`, `writing.qnk`, …) and switch between them by running `UNLOADKEY ALL ; LOADKBD <name>.qnk` at the CMline.

10.6 Persistence and startup

Every binding you make — whether via `LOADKEY` or via `LOADKBD` — is automatically saved to the config file alongside the rest of QuickNotes' settings. On the next launch, QuickNotes replays those bindings as part of startup. The order is:

1. Library auto-load (so a binding to a routine name resolves).
2. Replay saved key bindings from the config.
3. Run `startup.qns` if present.

The startup-after-replay ordering means your `startup.qns` can override the saved bindings if it wants to. The common pattern is to put `LOADKBD work.qnk` in `startup.qns` and let the profile file be the source for your bindings.

Where bindings live

The replayed bindings come from `key_bindings` in the JSON config file. UNLOADKEY ALL clears them in the running process, but if you also want to clear the config-stored copy, save an empty set afterwards: `UNLOADKEY ALL ; SAVEKBD empty.qnk` (and don't load that file later). Or edit the config file directly with QuickNotes closed.

10.7 A worked example

You write a lot of dated notes, so you want a "stamp" key that inserts today's date plus a separator. Open the user library (`EDITLIB`) and add:

```
SUB stamp
   # DOC: Insert today's date and a separator at the cursor.
    WRL $DA --
ENDSUB
```

Save, `RELOADLIB`, then bind it:

```
LOADKEY STAMP
```

Press `F2` when the dialog asks for the key. From now on, F2 inserts a line like `2026-05-11 --` followed by a newline, anywhere in the document. The binding is in your config, so tomorrow's F2 does the same thing.

Chapter 11

Startup and Configuration

This chapter covers what happens when QuickNotes launches — where it looks for files, what persists between sessions, and how `startup.qns` lets you arrange a familiar working environment every time you start.

11.1 Portable vs user-profile mode

QuickNotes runs in one of two modes, chosen at launch:

Portable mode (the default). All of QuickNotes' state lives next to `quicknotes.py`: the config file, the `notes/` folder, the `scripts/` folder. Drop the whole directory on a USB stick or into a synced cloud folder and your setup travels with you. This is what you'll be using unless you ask for otherwise.

User-profile mode (opt-in via `--user-profile`). State goes into your home directory: the config file at `~/.quicknotes.json`, default notes at `~/Documents/QuickNotes/`, default scripts at `~/Documents/QuickNotes/scripts/`. This was the original behaviour, and it's still available for users who prefer Unix-style hidden-dotfile convention.

Both modes accept a `--notes-dir DIR` override that uses `DIR` as the

notes folder *for this session only*, without writing the choice to the config. Handy for working on a project that lives elsewhere on disk.

11.2 Startup sequence

When you launch QuickNotes, the following happens in order:

1. **Window builds.** The main window, menu bar, and CMline strip are constructed but not yet shown.
2. **Config loads.** Settings from the config file are read and applied — fonts, line numbers, dark theme, tab width, recent files, key bindings, cmdline history, the CMline working directory, and so on.
3. **System library compiles.** The bundled system library (`HELLO`, `TIMESTAMP`, `WHEREAMI`) is parsed and registered.
4. **User library auto-loads.** If `scripts/user.qnu` exists, its routines are merged on top of the system library (last-wins on name collisions).
5. **Key bindings replay.** Saved bindings from `config_data["key_bindings"]` are re-applied app-wide.
6. **startup.qns runs.** If `startup.qns` exists in the directory that holds `quicknotes.py`, it's run as a script.
7. **Initial options apply.** Always-on-top, dark-theme toggle, etc. — the cosmetic state.
8. **Window shows; editor takes focus.** You're ready to type.

The ordering matters in a few places. The library auto-loads *before* key bindings replay, so a binding to a routine name (`LOADKEY QUOTE`) finds its target. Key bindings replay *before* `startup.qns`, so a startup script can override saved bindings via `LOADKEY` or `LOADKBD`. The startup script runs only after the GUI is fully on-screen, so a `MSG` or `INPUT` in `startup.qns` displays as a proper modal dialog rather than hanging.

11.3 startup.qns

`startup.qns` is QuickNotes' analog to XyWrite's `STARTUP.INT` script — a config-time script that runs once at every launch. The file must live in the same directory as `quicknotes.py` (*not* in `scripts/` — that folder is reserved for ad-hoc `.qns` scripts you `RUN` explicitly).

A typical `startup.qns` ties together several of the things earlier chapters covered:

```
QUIET

# Make sure the user-library knows our extra routines.
ADDLIB scratch.qnu

# Replace any saved key bindings with our profile-of-the-day
UNLOADKEY ALL
LOADKBD work.qnk

# Greet me by name.
SET who = $USER
MSG /T:2 Welcome back, $who!
```

`startup.qns` is just an ordinary script — if it fails, the error is reported on the status line but startup itself continues. You can edit it freely; any line that works in a `.qns` script works here. The status line shows the absolute path being read, so you'll always know which `startup.qns` ran.

11.4 Three independent fonts

QuickNotes maintains three completely independent font settings, each reachable from the **Options** menu and each with its own keys in the config file:

Setting	Affects	Config keys
Editor font	The note buffer you type into.	`font_family`, `font_style`, `font_size`
Preview font	The Markdown preview pane (cosmetic only).	`preview_font_family`, `preview_font_style`, `preview_font_size`
Styled-PDF font	Styled-PDF export prose (body, headings, lists).	`pdf_prose_family`, `pdf_prose_size`

The Styled-PDF font dialog lives under *Options* → *Styled-PDF* → *Font…* (alongside *Margins…*; see the next section). Each font dialog has Family / Style (where applicable) / Size widgets plus a live preview pane, and an **Apply** button that writes the chosen values to the config file immediately. Each setting is self-contained — changing one has no effect on the others.

A few practical notes:

- The **preview** font is purely visual. It does *not* affect either the plain-text PDF (which always uses Courier) or the styled PDF (which uses the Styled-PDF font). Use a comfortable reading face here regardless of what you want printed output to look like.
- The **Styled-PDF font** has a special "(use editor font)" option at the top of the family list, plus a *Use editor font* checkbox. Picking either falls back to the editor font (the default). Choose an explicit family — typically, a serif like Cambria, Georgia, or Charter — when you want to give your printable documents a finished look.
- Code blocks in the preview pane and in styled PDF output stay monospace (Consolas) regardless of the preview / Styled-PDF font, so fenced code keeps its alignment.
- The **editor** font is what tab-width measurement is computed against, so changing it triggers a re-measure across all open tabs. The other two don't.

11.5 Styled-PDF margins

Options → *Styled-PDF* → *Margins…* opens a small dialog with four entry fields (Top, Right, Bottom, Left) controlling the page margins of styled-PDF export. Each value accepts a number plus an optional unit suffix:

Suffix	Unit	Example
`cm`	centimetre	`1.5cm`
`mm`	millimetre	`12mm`
`in`	inch	`0.75in`
(none)	centimetre	`1.5`

The default is `1.5cm` on all four sides, matching the historical hardcoded value. Negative values and unrecognized input are rejected with a clear error message; Enter accepts, Escape cancels.

The values persist as config keys `pdf_margin_top`, `pdf_margin_right`, `pdf_margin_bottom`, and `pdf_margin_left`. Both the CSS embedded in the rendered HTML and the Chrome DevTools `printToPDF` call receive the same margins, so the body width the renderer plans for matches the printable area on paper — text won't reflow at the page edge regardless of which margin you change.

The page-number footer (if enabled via *Options* → *Page numbers in styled PDF*) inherits the left/right margins, so the page number stays inside the printable area at any setting.

11.6 Dark mode

Options → *Dark theme* flips QuickNotes to a low-light palette suitable for working in dim rooms or for users who simply prefer dark interfaces. The toggle re-themes the whole window: the three main panels (notes pane, editor, preview), the surrounding chrome (frames, search and tag entries, buttons, scrollbars, tab strip), the CMline, and the status line all switch to the dark

palette together. On Windows, the title bar across the top of the window also flips dark via the standard `DwmSetWindowAttribute` API; the window frame's resize border, the maximise / minimise / close buttons, and the title text all follow.

What's *not* themed (and why):

- The top-level **menu bar** (the File / Edit / Search / View / … strip across the top of the window) is OS-drawn on Windows and does not respect per-app dark-mode requests in any reliable way. The popup / cascade menus that drop down from those titles *are* themed, so once you open a menu you'll see it in dark colors. The menu bar strip itself stays in the system colors.
- Dropdown widget arrows and a few other native chrome accents follow whatever the OS does, since they're drawn outside Tk's control.

If the menu bar strip's lighter shade is distracting, the *Options → Auto-hide menu bar* feature (below) hides it entirely until you press Alt.

The dark palette persists (`dark_theme` config key) and restored on next launch. Toggling back to light is symmetric: the ttk theme reverts, the title bar repaints, and the chrome colors all snap back.

11.7 Auto-hide menu bar

Options → Auto-hide menu bar (Alt+Alt to toggle) turns the top-level menu bar into a tap-Alt-twice-to-show element. When the option is on, the bar is hidden at launch and on every dark-mode toggle. **Double-tapping Alt** (two complete press + release cycles within about 400 ms) toggles its visibility.

The double-tap gesture is deliberate: a single Alt press, or a second tap that comes too slowly, does nothing. Combined with the existing "Alt-alone" filter that ignores Alt presses inside multi-key chords (`Alt+F`, `Alt+Tab`, `Alt+Enter`, …), this keeps menu accelerators and OS-level Alt shortcuts from flickering the menu bar. A `<FocusOut>` reset prevents the toggle

from firing when you Alt+Tab away and back.

When the bar is hidden, the menu accelerators (`Alt+F`, `Alt+E`, …) cannot find a target menu — you have to press Alt first to make the bar visible. Once visible, accelerators behave normally and the bar stays open until you press Alt again or explicitly hide it.

The same visibility state is reachable from scripts and key bindings via three CMline verbs:

Verb	Effect
`SMB`	**S**how **M**enu **B**ar.
`CMB`	**C**lear (hide) **M**enu **B**ar.
`TMB`	**T**oggle **M**enu **B**ar visibility.

These change only the *current* state; they don't flip the persistent `auto_hide_menu_bar` config key, so your saved preference survives intact.

11.8 Sync preview scrolling

Options → *Sync preview scrolling with editor* keeps the Markdown preview pane's viewport aligned with the editor's. When you scroll the editor — via arrow keys, **PgUp / PgDn**, the mouse wheel, the scrollbar, or by typing past the bottom visible line — the preview re-scrolls to match. The setting is *on* by default. Turn it off if you want to scroll the two panes independently (e.g. to consult the rendered output of an earlier section while you edit further down).

The sync is proportional: it sends the editor's *top fraction* (how far down the document the top of the viewport is) to the preview. For prose-heavy notes the two stay closely aligned; for documents with many headings or long fenced code blocks the preview can drift a few lines from the source because headings render taller than their single source line and code blocks

lose their ``` fence lines in the preview. The drift resets every time you scroll back to the top.

When you toggle the preview pane on (`Ctrl+P`) while sync is enabled, the pane opens at the editor's current scroll position — no jarring snap to the top.

11.9 The config file

QuickNotes' persistent settings live in a JSON file:

- **Portable mode (default):** `<script-dir>/quicknotes.config.json`
- **User-profile mode:** `~/.quicknotes.json`

The file is written automatically whenever something settable changes, so you don't manage it directly under normal use. What's in it, in rough categories:

Group	Keys
Window	`geometry`
Editor font	`font_family`, `font_style`, `font_size`
Preview font	`preview_font_family`, `preview_font_style`, `preview_font_size`
Styled-PDF font	`pdf_prose_family`, `pdf_prose_size`
Styled-PDF margins	`pdf_margin_top`, `pdf_margin_right`, `pdf_margin_bottom`, `pdf_margin_left`
Display	`line_numbers`, `visible_returns`, `dark_theme`, `always_on_top`, `tab_width`, `cursor_style`
UI size	`ui_font_size`, `ui_scale`
Colors	`custom_fg`, `custom_bg`, `custom_fg_saved`, `custom_bg_saved`
Behaviour	`confirm_run_commands`, `persistent_selection`, `gui_cmdline`, `last_dt_format`, `auto_hide_menu_bar`, `sync_preview_scroll`
Folders	`folder` (notes folder), `scripts_dir`

Group	Keys
Recent / history	`recent_files`, `cmdline_history`, `cmdline_cwd`, `drive_cwds`
Key bindings	`key_bindings`
System library	`system_library` (defaults to true)

Deleting the file is harmless: QuickNotes recreates it on next launch with default values. Editing it by hand also works — close QuickNotes first, edit, save, relaunch. JSON syntax is enforced; a malformed file is reported on startup and the defaults are used for that session.

Disabling the system library

Set `system_library` to `false` in the config to suppress the bundled system library entirely. Useful if you want your own `HELLO` routine without the built-in one in the way, although the same effect can be had by defining the override in `user.qnu` (user routines beat system ones on name collision).

11.10 Saving and loading configuration snapshots

The active config file changes constantly as you work, which makes it a poor place to stash a setup you want to keep or share. For that, QuickNotes can write a *named* snapshot of the entire configuration to a `.json` file of your choosing, and load one back on demand. Reach these from *File* → *Save configuration…* and *File* → *Load configuration…*, or from the command line with `SaveCfg` and `LoadCfg`.

`SaveCfg <file>` writes the complete configuration — the same shape as the auto-saved config, including fonts, theme, styled-PDF margins, ev-

ery *Options* toggle, key bindings, recent files, command history, the CMline working directory, and the current window geometry. If `<file>` has no extension, `.json` is appended. An existing file is never clobbered without a confirmation prompt. The result is a portable profile you can copy to another machine or keep as a named "look".

`LoadCfg <file>` reads such a snapshot back and applies it immediately — no restart. The theme, editor and preview fonts, PDF margins, tab width, line spacing, cursor, line numbers, visible returns, always-on-top, the notes pane, the menu bar, every *Options*-menu checkbox, your key bindings, and the window size and position all change live. Two settings are exceptions: **UI scale** and **UI font size** can only fully change when QuickNotes starts, so `LoadCfg` updates their stored values and notes in the status line that a restart is needed to complete them. The loaded snapshot is merged into the active config and persisted, so the new look survives the next launch regardless.

A loaded snapshot does **not** become the active config file: QuickNotes keeps reading from and writing to its normal config location. To check which file is which from a script, read `$CF` (the active config file's path) and `$CFL` (the path of the most recent file passed to `LoadCfg` this session; empty until you load one).

Profiles of the day

Pair snapshots with 'startup.qns' or a key binding for one-keystroke profile switching: 'LOADKEY LoadCfg night.json' binds a key that flips your whole look to a saved "night" profile, the same way '.qnk' files do for key bindings alone.

11.11 The folder layout

Once QuickNotes has run at least once in portable mode (the default), the directory containing `quicknotes.py` looks something like this:

```
quicknotes/
   quicknotes.py
   quicknotes.config.json
  startup.qns            (optional -- your launch script)
  work.qnk              (optional -- key-binding profile)
   writing.qnk              (...and any others)
   notes/
      Things to remember.md
      Project notes.md
   scripts/
      user.qnu                  (your library)
      stamp-date.qns            (your scripts)
      count-todos.qns
```

Three kinds of file live alongside `quicknotes.py` itself:

- The JSON config file.
- `startup.qns`, the launch-time script.
- `.qnk` key-binding profile files. Following XyWrite's convention, these are configuration — part of how you've set the program up — and they sit at the top level rather than in `scripts/`.

The `notes/` folder is where new notes land by default. You can override it per-session with `--notes-dir DIR`, or change the default by saving a note to a different folder via *File* → *Save As* (which writes `folder` to the config).

The `scripts/` folder holds two kinds of file:

- `.qns` scripts that the *Scripts* menu auto-populates and that `RUN <name>` invokes by basename.
- `.qnu` user libraries that `LOADLIB` / `ADDLIB` parse for routines.

A `.qns` file you create here shows up in the *Scripts* menu the next time the menu opens. No reload step required.

11.12 Where to from here

That's the full QuickNotes tour. The appendices cover the finer-grained references — every command in alphabetical order, every special variable with its value, the full string-function reference, complete specifications of the `.qnu` and `.qnk` file formats, and a small gallery of useful scripts to seed your own library.

Appendix A

Command Index

Every CMline verb QuickNotes recognizes, in alphabetical order. Each entry points back to the chapter or section where the verb is covered in detail. Aliases are listed under their canonical name.

Verb	What it does	See
`?`	About dialog.	7.25
`AB`	Close active tab without verifying. Alias: `AB/NV`.	4.4, 7.1
`AB/NV`	Alias for `AB`.	4.4, 7.1
`AB/V`	Close active tab; if modified, prompt to save / discard / cancel.	4.4, 7.1
`ABD`	Delete the active note (no confirmation) and close its tab.	7.1
`ABOUT`	Same as `?`.	7.25
`ADDLIB`	Additively merge a `.qnu` into the current library.	9.3
`APPEND`	Append selection to the clipboard.	7.14
`AUTOSAVE [ON|OFF]`	Toggle, set, or report the AutoSave flag (mirror of *Options* → *AutoSave*). Bare `AUTOSAVE` reports the current state.	4.5
`BC`	Like F5: blank + summon + focus the CMline; optional text is staged.	6.3
`BD`	Backspace — delete the character before the cursor.	7.4
`BF`	Bottom of file.	3.1, 7.2
`BREAK`	Exit the innermost enclosing `WHILE` loop.	8.9
`BS`	Cursor to bottom of visible screen.	7.2

Verb	What it does	See
CA	Open <path> in a tab, editable. Alias: ED, EDIT.	4.4, 7.1
CALL	Invoke a subroutine.	8.14
CC	Swap case of selection (or char under cursor).	7.5
CCL	Clear Caps Lock.	3.4, 7.7
CD [path]	Show or change the CMline working directory.	6.11, 7.26
CD/D <drive:path>	Change drive and directory in one step (Windows).	6.11, 7.26
CGC	Clear (turn off) the GUI command line.	6.2
CH, CHA, CI, CIA	Replace every match. CH/CHA = CI/CIA aliases.	5, 7.11
CHARCOUNT	Status-line character count of selection (or whole document).	7.15
CIN	Clear insert mode (= overstrike).	3.3, 7.6
CL	Cursor left with wrap.	7.2
CLERR	Clear $ERROR.	8.15
CLIP <text>, CY <text>	Set clipboard to <text> (literal rest of line, with $-sub and escape handling). Bare CLIP clears the clipboard.	7.14
CMB	Clear (hide) the menu bar.	7.9, 11.7
CNL	Clear Num Lock.	7.7
CNP	Clear (hide) the notes pane.	4.1, 7.10
COMMANDS	Open the in-app Command Reference window.	7.25
CONTINUE	Restart the innermost enclosing WHILE loop.	8.9
COPY <src> <dst>	Copy a file on disk; prompts before overwriting an existing destination.	7.26
COPY/NV <src> <dst>	Like COPY, but overwrites silently.	7.26
CP	Copy the selected text to the cursor position.	7.3
CPS	Clear persistent selection.	3.5, 7.8
CR	Cursor right with wrap.	7.2
CSL	Clear Scroll Lock.	7.7
CV, CVA	Walk match-by-match through replacements (Change-Verify).	5, 7.11
D:	Switch CMline working directory to drive D (Windows).	4.4, 7.1

Verb	What it does	See
DA	Insert the current date / time at the cursor.	7.27
DEL <path>	Delete a file on disk; prompts to confirm.	7.26
DEL/NV, ERNV	Like DEL, but skips the confirmation prompt.	7.26
DF	Define — start (or end) a selection anchor.	3.2, 7.3
DFA	Define all — select the whole document.	3.2, 7.3
DIR, LS	Scrollable directory listing; Enter (or double-click) edits, Ctrl+Enter opens/launches, Ctrl tap copies the highlighted path to the clipboard, Del deletes (with confirm); type-to-find and F3-find-again hints ride in the title bar. Accepts Windows-DIR-style switches.	7.26
DL	Define line.	3.2, 7.3
DN	Delete the current selection without copying it to the clipboard (or, with no selection, delete the character under the cursor).	7.4
DO	Display on: cancel DX and flush pending updates.	7.19
DP	Define paragraph.	3.2, 7.3
DS	Define sentence.	3.2, 7.3
DW	Define word.	3.2, 7.3
DWONLY	Like DW but always single-word.	7.3
DX	Display off: suppress screen refreshes.	7.19
ED	Alias for CA.	4.4, 7.1
EDFONT [n]	Set or show the editor (text-area) font size in points.	7.20
EDIT	Alias for CA.	4.4, 7.1
EDITLIB	Open the user.qnu file in a new tab.	9.3
EI	Alias for ENDIF.	8.8
ELSE, ELSE IF, ELSEIF	Conditional block continuations.	8.8
ENDIF	Close an IF block.	8.8
ENDSUB	Close a SUB block. Alias: ES.	8.14
ENDWHILE	Close a WHILE block. Alias: EW.	8.9
ENV	Copy an environment variable into $<var>.	7.18
ES	Alias for ENDSUB.	8.14
ET	Insert the elapsed-time string at the cursor.	7.27
ETS	Show the elapsed-time string on the status line.	7.27
EW	Alias for ENDWHILE.	8.9
EX	Alias for EXIT.	8.14

Verb	What it does	See
EX1	Alias for EXITALL.	8.14
EXISTS	Test path existence; store 1 or 0.	7.18
EXIT	Exit current sub; if at script top level, end the script.	8.14
EXITALL	Exit every running script back to the CMline.	8.14
GH	Go to CMline (focus the command line, no clearing).	7.24
GOTO	Jump to a :label.	8.10
GT	Go to text (focus the editor pane).	7.24
HELP	Open the in-app Usage / Quick Reference window.	7.25
HISTORY [filter]	Popup of CMline history; optional <n> or "<substring>" filter. Enter re-dispatches the picked line; Del removes it from the history.	6.8
HM	Cursor to top of visible screen.	7.2
IF	Conditional execution (single-line or block).	8.8
IFNOT	Shorthand for IF NOT.	8.8
INPUT	Modal entry dialog into $<var>.	7.17
JMP	Jump to character offset *n*.	7.2
LB	Line beginning.	3.1, 7.2
LC	Lowercase the selection (or char under cursor).	7.5
LD	Line down.	7.2
LE	Line end.	3.1, 7.2
LEADING [n]	Set or show editor / preview line spacing as a percent of the editor font's line height. Clamped 0..200. Alias: LSP.	7.20
LISTKEYS	List current key bindings.	10.3, 7.25
LISTLIB	List loaded library routines with their DOC strings.	9.3, 7.25
LISTVAR	Pop a TXT dialog listing every script variable in memory.	8.1
LL	Left without wrap.	7.2
LN	Toggle the line-number gutter.	7.28
LoadCfg	Load a configuration snapshot and apply it live.	4.4, 11.10
LOADKBD	Merge bindings from a .qnk file.	10.5
LOADKEY	Bind a command to an available key.	10.1
LOADLIB	Load (or reload) the user library.	9.3

Verb	What it does	See
LOCAL	Declare one or more sub-scoped variables (LOCAL i j tmp).	8.14
LR	Right without wrap.	7.2
LSP [n]	Alias for LEADING.	7.20
LU	Line up.	7.2
MD	Move text Down on screen one line (reveals line above); cursor doesn't move.	7.2
MERGE, ME	Insert a file's contents into the active note at the cursor. Replaces the current selection, if any.	7.26
MERGE/NV, ME/NV	Like MERGE, but leaves the cursor at its original position after the insert.	7.26
MSG	Modal info dialog.	7.17
MU	Move text Up on screen one line (reveals line below); cursor doesn't move.	7.2
MV	Move the selected text to the cursor position (delete from origin, insert at cursor).	7.3
NC	Next character (synonym of CR).	7.2
NE	New note.	4.4, 7.1
NL	Start of next line.	7.2
NP	Next paragraph.	7.2
NS	Next sentence.	7.2
NW	Next word.	3.1, 7.2
NX	Next tab.	7.1
OF [n]	Set or show the editor / preview margin offset, in character columns of the editor font.	7.20
PC	Previous character (synonym of CL).	7.2
PD	Page down.	7.2
PDF	Export the active note as a plain-text PDF (monospaced).	7.28, 11.5
PDFS, PDFSTYLED	Export the active note as a styled PDF (Markdown-rendered). Asynchronous; compatible with WAIT.	7.28, 11.5
PL	Start of previous line.	7.2
PP	Previous paragraph.	7.2
PR	Print text to the status line.	7.17
PREPEND	Prepend selection to the clipboard.	7.14
PRINTS, RV, TYS	Toggle the Markdown preview pane. (The command RV is distinct from the special variable $RV.)	7.28

Verb	What it does	See
PS	Previous sentence.	7.2
PU	Page up.	7.2
PW	Previous word.	3.1, 7.2
PX	Previous tab.	7.1
QUIET [ON|OFF]	Suppress informational status echoes. Bare = ON; OFF re-enables.	8.16
QUIET/NV [ON|OFF]	Suppress ALL status output (info AND error); only fatal abort dialogs survive.	8.16
QUIT	Close the window; prompts to save / discard / cancel per modified tab.	7.25
QUIT/NV	Close the window without prompting; unsaved changes are discarded silently.	7.25
RC	Rub out (forward-delete) the character under the cursor, like the Delete key.	7.4
RD	Delete the current selection.	7.4
RE	Open <path> read-only. Alias: READ.	4.4, 7.1
READ	Alias for RE.	4.4, 7.1
READFILE, RF	Read file contents into $<var>.	7.13
READKEY <var> [prompt], RK <var> [prompt]	Pause for one keystroke; store its character / keysym in $<var> and its scan code in $SC.	7.29
READKEY/NV <var>, RK/NV <var>	Silent variant of READKEY: no dialog, no prompt. Captures one keystroke into $<var> and $SC. Use PR beforehand if a visual cue is wanted.	7.29
READWRITE	Alias for RW.	4.4, 7.1
REC, RECALL	Discard edits made since the last manual save and revert the active tab to that snapshot.	4.5, 7.1
REDO	Redo the last undone change (optional [n]).	7.23
RELOADLIB	Replay the full LOADLIB + ADDLIB sequence.	9.3
REN	Alias for RENAME.	7.26
REN/NV	Alias for RENAME/NV.	7.26
RENAME <old> <new>	Rename <old> to <new> in the same directory. <new> must be a bare filename. Refuses directories. Prompts before overwriting. If <old> is open in a tab, the tab follows the rename in place.	7.26
RENAME/NV <old> <new>	Like RENAME, but overwrites an existing destination silently.	7.26

Verb	What it does	See
RETURN	Return from a subroutine, optionally setting $RV.	8.14
RF	Alias for READFILE.	7.13
RL	Delete the rest of the line.	7.4
RS	Delete the current sentence.	7.4
RUN	Run a .qns script by name.	8.12
RUNHERE	Run the fenced QPL code block under the cursor.	8.12
RW	Toggle the active tab's read/write state.	4.4, 7.1
SA	Save the active tab.	4.4, 7.1
SAAS	Save as; no path opens a dialog.	4.4, 7.1
SAD	Save selected text to a new file.	4.4, 7.1
SaveCfg	Save the full configuration to a .json snapshot.	4.4, 11.10
SAVEKBD	Save current bindings to a .qnk file.	10.5
SCL	Set Caps Lock.	7.7
SE, SEA, SEB, SEBA	Search (forward, backward, case-sensitive variants).	5, 7.11
SET	Assign a variable.	8.1
SETERR	Set $ERROR.	8.15
SF	Save and close every open tab.	4.4, 7.1
SGC	Set (turn on) the GUI command line.	7.1
SIN	Set insert mode.	3.3, 7.6
SL	Save all open tabs.	4.4, 7.1
SMB	Show the menu bar.	7.9, 11.7
SNL	Set Num Lock.	7.7
SNP	Show the notes pane.	4.1, 7.10
SORT	Sort the lines of the current selection.	7.22
SOV	Set overstrike mode.	3.3, 7.6
SPLIT	Split a string around a delimiter into a numbered array.	7.16
SPLITS	Like SPLIT but the delim is a set of characters.	7.16
SPS	Set persistent selection.	3.5, 7.8
SSL	Set Scroll Lock.	7.7
SUB	Begin a subroutine definition.	8.14
SV	Save the current selection to a named script variable.	8.1
TCL	Toggle Caps Lock.	7.7

Verb	What it does	See
TF	Top of file.	3.1, 7.2
TGC	Toggle the GUI command line.	7.1
TIN	Toggle insert/overstrike. Alias: TOV.	3.3, 7.6
TMB	Toggle menu-bar visibility.	7.9, 11.7
TNL	Toggle Num Lock.	7.7
TNP	Toggle the notes pane.	4.1, 7.10
TOV	Alias for TIN.	3.3, 7.6
TPS	Toggle persistent selection.	3.5, 7.8
TSL	Toggle Scroll Lock.	7.7
TXT	Scrollable monospace dialog.	7.17
UC	Uppercase the selection (or char under cursor).	7.5
UIFONT [n]	Set or show the UI font size in points.	7.20
UIRESET	Restore UI font size and Tk scaling to their defaults.	7.20
UISCALE [factor]	Set or show the Tk scaling factor; restart for full effect.	7.20
UNDO	Undo the last change (optional [n]).	7.23
UNLOADKEY	Remove a key binding.	10.4
UNLOADLIB	Drop every user-library routine.	9.3
VR	Toggle the visible-returns gutter (pilcrow ¶ at each hard newline).	7.20
WAIT [ms]	Pause script execution. Bare WAIT blocks for the previous async command to complete; with ms, sleeps that many milliseconds.	7.29
WHILE	Loop.	8.9
WORDCOUNT	Status-line word count of selection (or whole document).	7.15
WR	Insert text at cursor. Alias: WRITE.	7.12
WRITE	Alias for WR.	7.12
WRITEFILE, WF	Write $<var> to a file; overwrites silently if it exists.	7.13
WRITEFILE/V, WF/V	Like WRITEFILE, but prompts before overwriting an existing file.	7.13
WRITELN	Alias for WRL.	7.12
WRL	Like WR but append a trailing newline.	7.12
X	Run shell command; exit code in $RC.	6.10, 7.21

Verb	What it does	See
X/NV	Launch shell command and return immediately; output discarded. $RC is 0 on launch, -1 on launch failure.	6.10, 7.21
XC	Execute the command currently on the bottom-strip CMline.	6.3
XD	Cancel selection. Alias: YD.	3.2, 7.3
XV	Run shell command; capture stdout into $<var>.	6.10, 7.21
XY	Run shell command; yank stdout into buffer at cursor.	6.10, 7.21
YD	Alias for XD.	3.2, 7.3
ZT	Reset the elapsed-time stopwatch to zero.	7.27

Appendix B

Special Variables

QPL's read-only special variables, alphabetical by name. The `SET` command rejects writes to any of these. Their values reflect QuickNotes' current state at the moment the variable is read.

Name	Value
`$#`	Count of arguments passed to the current sub or script.
`$*`	All positional arguments, joined with single spaces.
`$1` … `$#`	Individual positional arguments to the current sub or script.
`$AU`	`1` if the user-toggleable AutoSave is enabled, else `0`.
`$CB`	Current clipboard contents (4 KB cap).
`$CC`	Replace-count from the most recent `CH` / `CHA` / `CI` / `CIA` / `CV` / `CVA`. A not-found run resets it to `0` and trips `$ERROR`.
`$CF`	Absolute path of the active configuration file — the JSON QuickNotes reads at startup and rewrites on every option change (and after a `LoadCfg` merge).
`$CFL`	Absolute path of the most recent file loaded via `LoadCfg` this session. Tracks the source snapshot, not the active config file (`$CF`); empty until the first successful `LoadCfg`.
`$CH`	Character under the cursor (the char to the right of the insertion point). Empty at end-of-file.
`$CL`	`1` if Caps Lock is on, else `0` (Windows; `0` elsewhere).
`$CP`	Cursor's absolute character offset in the document (0-based).
`$CT`	1-based index of the active tab.
`$CWD`	CMline working directory.

Name	Value
$CX	Cursor's column on the current line (0-based).
$CY	Cursor's visual row within the visible viewport.
$DA	Today's date as ISO 8601, e.g. 2026-05-11.
$DF	1 if a selection is active, else 0.
$DS	Character offset of the selection's start (0-based). Falls back to the cursor offset ($CP) when no selection is active.
$DN	Character offset of the selection's end (0-based). Falls back to the cursor offset ($CP) when no selection is active.
$ED	Absolute path of quicknotes.py itself.
$EDFONT	Current editor font size in points (EDFONT command, font_size config key).
$ERROR	1 if the last error-aware command failed, else 0.
$ET	Elapsed seconds since the last timer reset.
$FB	1 if cursor is at the beginning of the file, else 0.
$FE	1 if cursor is at the end of the file, else 0.
$FI	Active note's filename (no directory).
$FP	Active note's full path.
$FS	Size of the active document in characters.
$GC	1 if the GUI command line option is on, else 0.
$LC	Total line count of the active document.
$LN	1-based line number the cursor is on.
$LS	Last status-line message string.
$MB	1 if the menu bar is currently displayed, else 0.
$MO	1 if the active tab has unsaved modifications, else 0.
$NL	1 if Num Lock is on, else 0 (Windows; 0 elsewhere).
$NP	1 if the notes pane is currently shown, else 0.
$OF	Editor / preview margin offset, in character columns of the editor font. 0 = flush to the widget edges. Set by the OF command or *Options → Margin offset…*.
$OS	Host OS: windows, macos, linux, or the raw platform string.
$PA	CMline working directory (synonym of $CWD).
$PS	1 if persistent-selection mode is on, else 0.
$QNDIR	Directory containing quicknotes.py (no trailing separator). Compose paths with $SLASH: "$QNDIR" . "$SLASH" . "scripts".
$RC	Last shell command's exit code.
$RE	1 if the active tab is read-only, else 0.

Name	Value
$RV	Last sub or library routine's return value.
$RL	1 if Scroll Lock is on, else 0 (Windows; 0 elsewhere). Named with R because $SL is already taken.
$SC	Scan code (Tk event.keycode, the platform virtual-key code on Windows) of the most recent key captured by READKEY / RK. 0 before any READKEY has fired in the current session.
$SE	Current selection text (4 KB cap).
$SF	Absolute path of the scripts/ folder.
$SLASH	Platform path separator: \ on Windows, / on macOS / Linux.
$SL	Currently selected text length in characters.
$TB	Current tab width in spaces.
$TC	Total number of open tabs.
$TM	Current time of day as HH:MM:SS.mmm (with milliseconds).
$TX	1 if the editor has focus, 0 if the CMline does.
$UIFONT	Current UI font size in points (UIFONT command, ui_font_size config key).
$UISCALE	Current Tk scaling factor (UISCALE command, ui_scale config key).
$VE	QuickNotes revision string (matches the cover-page date).
$VR	1 when Visible Returns is on, 0 when off (VR command, visible_returns config key).
$WC	Total word count of the active document (whitespace-delimited tokens).

Appendix C

Function Reference

The functions available inside QPL expressions — that is, in the right-hand side of `SET name = expr`, the condition of `IF` / `IFNOT` / `ELSEIF`, the condition of `WHILE`, and anywhere else an expression appears. All function names are case-insensitive. String arguments may be string literals (`"..."`), `$NAME` references, nested function calls, or any expression that evaluates to a string.

Function	Returns
`LEN(s)`	Character count of `s`.
`UPPER(s)`	`s` in upper case.
`LOWER(s)`	`s` in lower case.
`STRIP(s)`	`s` with leading AND trailing whitespace removed.
`LSTRIP(s)`	`s` with leading whitespace removed.
`RSTRIP(s)`	`s` with trailing whitespace removed.
`TRIMLEFT(s, n)`	`s` with the first `n` characters removed.
`TRIMRIGHT(s, n)`	`s` with the last `n` characters removed.
`SUBSTR(s, i, n)`	`n`-character substring of `s` starting at 0-based offset `i`.
`INDEX(s, sub)`	0-based index of the first occurrence of `sub` in `s`, or `-1` if absent.
`CONTAINS(s, sub)`	`1` if `sub` appears anywhere in `s`, else `0`.
`REPLACE(s, find, repl)`	All non-overlapping occurrences of `find` in `s` replaced with `repl`.

Function	Returns
`CHR(n)`	Single-character string for ASCII / Latin-1 code point `n` (0..255). Returns `""` for out-of-range or non-numeric `n`. Companion to `ASC`.
`ASC(s)`	ASCII / Latin-1 code point of the first character of `s`, as a decimal string. Empty `s` returns `"0"`. Companion to `CHR`.
`SEL()`	Current selection text. Equivalent to `$SE` but returned as a proper string value (safer when the selection contains characters that would mis-parse if substituted into the source).
`VAR(name)`	Value of `$<name>`, returned as a proper string. The function-call form of `$`-substitution.

`SEL()` and `VAR()` exist because `$`-substitution happens at the source-line level: if a substituted value contains `"`, `\`, or a control character, it can mis-parse downstream. `SEL()` and `VAR()` return values directly to the expression evaluator, bypassing the source-line lexer.

Appendix D

The .qnu Format

A `.qnu` file is QuickNotes' user-library format. It's plain UTF-8 text, parsed line-by-line into a set of named subroutines.

Grammar, informally:

```
file      := [frontmatter] (routine | blank-or-comment)*
frontmatter := "---" NL
               (any-line)* until
               "---" NL
routine     := "SUB" SP name NL
               (body-line)*
             "ENDSUB" NL    ("ES" is a synonym)
body-line  := comment | doc-line | any QPL command line
doc-line    := "#" SP "DOC:" SP text NL
comment     := "#" (SP text | NL) NL
```

Notes:

- The optional `---`-delimited frontmatter is tolerated and ignored, so a `.qnu` saved as if it were a note with title / tags / created header still parses.
- Outside `SUB` … `ENDSUB` blocks, everything is ignored. Blank lines

and comments at file scope are fine.

- A `# DOC:` line inside a routine sets that routine's documentation string. Multiple `# DOC:` lines concatenate with newlines.
- Any other `#` line inside a routine body is a QPL comment and passes through to the script runner.
- Routine names follow C-like identifier rules and are case-insensitive.
- The same name may not be used twice within a single file. Across `LOADLIB` + multiple `ADDLIB` calls, the last-loaded routine wins on name collision.

Built-in shadowing. The loader rejects any routine whose name collides with a built-in CMline verb (`SE`, `CH`, `SET`, etc.) or with a special variable name (`SE`, `ERROR`, `OS`, …). Rejected routines are reported on the status line; the rest of the file loads normally.

A minimal complete example:

```
SUB hello
   # DOC: Print a friendly greeting to the status line.
    PR Hello, world!
ENDSUB

SUB double
   # DOC: Double a number passed as an argument.
    RETURN $1 * 2
ENDSUB
```

Appendix E

The .qnk Format

A `.qnk` file is QuickNotes' key-binding profile format. It's plain UTF-8 text, one binding per line.

Grammar:

```
file        := (binding | blank-line | comment)*
binding      := key-spec WS+ command NL
comment      := "#" (text) NL
key-spec     := "<" modifier* keyname ">"
modifier    := "Control-" | "Shift-" | "Alt-"
```

Notes:

- The *key-spec* is a Tk-style binding string. The forms QuickNotes accepts as the `<...>` content are documented in chapter 10 ("What can you bind?"). Examples: `<F2>`, `<Shift-F7>`, `<Control-Shift-W>`, `<Alt-b>`.
- The *command* is everything after the key-spec, up to end of line, after one or more whitespace characters. The command may contain `;` chains and `$NAME` references — these are preserved verbatim and substituted at key-press time, not at load time.
- `#` starts a line comment at the beginning of a line (after any leading

whitespace) and runs to end-of-line. Mid-line `#` is preserved (so a binding like `PR Issue #4 fixed` still works).

- Blank lines are ignored.
- Spec/command pairs whose key-spec is reserved by QuickNotes itself (e.g. `<F1>`, `<F5>`) are skipped with a warning when `LOADKBD` reads the file, but other bindings in the same file still load.

A minimal example:

```
# Profile: daily work
<F2>                HELLO
<Alt-b>             QUOTE
<Ctrl-Shift-T>      TIMESTAMP

# Run a multi-step cleanup on Shift+F1
<Shift-F1>          DX; CH/T /\t/ /; DO
```

Appendix F

Sample Scripts

A small gallery of useful scripts to seed your own collection. Drop any of these into your `scripts/` folder as `.qns` files, or into your `user.qnu` as `SUB` … `ENDSUB` routines.

F.1 Insert a timestamp

A one-line stamp suitable for log entries:

```
WRL $DA $TM --
```

Save as `stamp.qns` in `scripts/`. Run with `RUN stamp` or bind to a key with `LOADKEY RUN stamp`. Inserts a line like `2026-05-11 14:23:07 --` at the cursor.

F.2 Word count of a selection

`CHARCOUNT` and `WORDCOUNT` are built-ins, but a script-friendly version that returns the count in `$RV` and is silent on no-op:

```
SUB wc
   # DOC: Word count of the current selection, into $RV.
```

```
    IF $SE == ""
        PR wc: nothing selected.
        SETERR
        RETURN 0
    ENDIF
    SET text = SEL()
    SET n = 0
    SET parts = SPLITS text " \t\n"
    IF $ERROR == 0
        SET n = $parts[0]
    ENDIF
    RETURN $n
ENDSUB
```

Library version (put in `user.qnu`). Call from the CMline: `WC; PR $RV words`.

F.3 Wrap selection in delimiters

A general "wrap the selection in something" helper:

```
SUB wrap
   # DOC: Wrap the current selection in $1 (open) and $2 (cl
    IF $SE == ""
        PR wrap: nothing selected.
        SETERR
        RETURN
    ENDIF
    LOCAL open close text
    SET open = $1
    SET close = $2
    SET text = SEL()
    DN
```

```
    WR $open
    WR $text
    WR $close
ENDSUB
```

Then:

```
WRAP " "
WRAP < >
WRAP "/* " " */"
```

F.4 Clean up trailing whitespace

A document-wide trim of trailing spaces and tabs at end of every line:

```
QUIET
CH/T /\t\n/\n/
SET passes = 0
WHILE 1
    CH/T / \n/\n/
    IF $ERROR BREAK
    SET passes = $passes + 1
ENDWHILE
PR Cleaned $passes passes.
```

A few notes on this one. The first `CH` converts tab+newline to just newline (one pass — tabs at line-end are rare). The loop then repeats space+newline -> newline until no more matches; the loop handles multiple trailing spaces on a single line (each pass strips one space). ‘

Appendix G

Markdown Support

QuickNotes renders Markdown in two places: the **preview pane** (toggled with **Ctrl+P** or *View* → *Toggle Markdown preview*) and **PDF export** (via `PDF` / `PDFS` from the CMline, or *File* → *Export*). Both use the same parser, so what appears in the preview is what ends up in the PDF.

The parser is intentionally compact — it covers the constructs most note-taking workflows lean on, without the surface area of full CommonMark or GitHub Flavored Markdown. This appendix lists what is supported and what is not, so you can write notes with predictable results.

G.1 Block-level constructs

Markdown	Renders as
`# Heading 1` through `###### H6`	Headings, levels 1-6 (`<h1>`-`<h6>`)
`---`, `***`, `___` (3+ chars)	Horizontal rule (`<hr>`)
` ```lang ... ``` `	Fenced code block (see *Code blocks*)
`- item`, `* item`, `+ item`	Bullet list (`<ul>`)
`1. item`, `1) item`	Numbered list (`<ol>`)
`- [ ] task`, `- [x] task`	Task-list checkbox (see *Lists*)
2-column indent under a list item	Nested list level

Markdown	Renders as
`> quoted` (repeat > for nesting)	Blockquote (`<blockquote>`)
Trailing two spaces at end of line	Hard line break (` `) inside paragraph
Blank line	Paragraph break

All six heading levels are recognized. The PDF/HTML renderer shrinks h4-h6 toward body size so the visual hierarchy stays clear without growing past h3; the preview pane does the same.

Blockquotes accept any block-level content inside (including nested blockquotes, headings, and lists). A blockquote ends at the first blank line or at a line that doesn't start with >.

G.2 Inline constructs

Markdown	Renders as
`**text**`	**text** (`<strong>`)
`*text*` or `_text_`	*text* (`<em>`)
`~~text~~`	~~text~~ (`<del>`, line-through)
`` `text` ``	`text` (`<code>`)
`[label](url)`	Hyperlink with `label` as visible text
Bare `http://...`	Auto-linked URL
Bare `https://...`	Auto-linked URL

Inline markup combines in the obvious ways (e.g. a `**bold**` phrase inside a paragraph), but the parser doesn't validate nesting rigorously; pathological inputs may produce surprising output.

G.3 Lists and task items

Bullet lists open with −, *, or +. Numbered lists open with `1.` or `1)`. Nesting uses 2-column indentation steps (tabs count as 2 columns).

Bullet items may carry a GitHub-style task checkbox by prefixing the content with `[ ]` (unchecked) or `[x]` / `[X]` (checked). The renderer replaces the marker with a Unicode ballot box and an empty-or-checked variant:

```
- [ ] Buy milk
- [x] File taxes
- [ ] Schedule dentist
```

renders as a bullet list where each item begins with an empty ballot-box glyph (U+2610) for `[ ]` items or a checked ballot-box glyph (U+2611) for `[x]` items. The preview pane and styled-PDF export use the same Unicode characters.

Numbered lists ignore the `[ ]` / `[x]` prefix even if present (GitHub's spec only recognizes tasks on bullet lists).

G.4 Code blocks and syntax highlighting

Fenced code blocks open with three backticks and an optional language tag, and close with three backticks:

````
```python
def hello():
    print("hello")
```
````

Recognized language tags (canonical name and common aliases):

Canonical	Aliases
`autoit`	`au3`, `a3x`
`bash`	`sh`, `shell`, `zsh`, `bat`, `cmd`, `batch`
`c`	`h`
`cpp`	`c++`, `cxx`, `cc`, `hpp`, `hxx`
`csharp`	`cs`, `c#`
`javascript`	`js`, `ts`, `typescript`, `node`, `jsx`, `tsx`
`json`	—

Canonical	Aliases
`python`	`py`, `python3`
`sql`	`psql`, `mysql`, `postgres`
`zig`	—

Unknown or absent tags (```` ``` ```` alone, or e.g. ```` ```rust ````) render as plain monospace with no semantic coloring. The label is preserved in the output, so if a future release adds the language, the same source will pick up highlighting without a rewrite.

G.5 Constructs NOT supported

Common Markdown features that QuickNotes does not recognize. Lines using these constructs render as plain text:

- Tables (`| col | col |` with `|---|---|` separator)
- Images (`![alt](url)`)
- Reference-style links (`[label][1]` … `[1]: url`)
- Footnotes (`[^1]` and `[^1]: ...`)
- Inline HTML pass-through (HTML tags are escaped and shown literally)
- Definition lists
- Setext-style underline headings (=== or −−− under a text line — only the #-prefix form is recognized)

For notes that need any of these (a memo with a real table, say), keep the source in plain Markdown and use an external converter (pandoc or the like) for the final formatted version. The QuickNotes source file stays untouched and remains portable.