



THE FIRST PAIR EMACS READER

DANTE • INFO • 1UNIX • iOS

Info

Inferno

Nel mezzo
del cammin
di nostra
vita, mi ritrovai
per una selva
oscura...

Purgatorio

Per quindi
uscimmo
a riveder
le stelle.



- ★ Top
- ★ Inferno
- ★ Purgatorio
- ★ Paradiso



info Top
(dir)

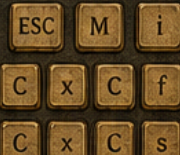
- * Menu:
- * Dante
- * Inferno
- * Purgatorio
- * Paradiso
- * Notes

-- Node: Top --
iSH-\$ emacs

Paradiso

L'amor che
move il sole
e l'altre
stelle.

E



The First Pair Emacs Reader Book

Parallel Texts, Living Dictionaries, and Touch in GNU Info — from the Laptop to the iPhone

Alexy Khrabrov

2026-09-03

Contents

Preface	3
Why Info	4
What a parallel reader has to do	4
The Bundle on Disk	6
Node names	6
Why the builder writes the bytes	7
The manifest and the sealed inventory	7
Two Manuals, Three Windows	8
Redirecting a cross-reference	9
Making room	9
Every Word Answers	11
Marked words and where to find them	11
Aligned editions: every source word	12
Walking the source	13
The current word stays underlined	14
The lexicon: normalise, analyse, gloss	14
The compact dictionary	15
Translations as State	17
Where it started	17
The model that stuck	17
Checkbox menus	18
Feedback that survives the menu	20
The header row: tap to hide, tap to reorder	20
Tr< and Tr>: swap, not cycle	21
2nd: the favorites button	22
What is left of the old commands	23
Touch	24
What a terminal tells Emacs about a finger	24
A button is a propertized string	24
The command runs in the window that was tapped	26
The three bars	27
Single keys	29
Tapping the text	30
Scrolling with a finger	30

What the reader cannot fix	31
Two Bars, One Contract	32
The wrong turn	32
Why keys and not mouse reports	33
Long press as the inverse	33
Toggling the native bar off	33
The contract, in one table	34
Big Books on Small Machines	35
The nine-megabyte file	35
Regions by byte offset	37
Sharded tables	37
The numbers	38
Reordering in place	38
Resume, Package, Verify	41
Where you were	41
Which reader runs	42
The package and its sidecar	42
Proving it works: <code>firstpair-check</code>	43
Testing Emacs from Python	44
1Unix	45
A finger becomes a mouse	45
DECSET 1003, which <code>hterm</code> did not have	45
The atomic tap	46
A drag becomes a wheel	47
The keyboard that would not stay down	47
The native strip, revisited from this side	48
Fonts, cells, and first paint	48
The phone check	48
What the fork is and is not	49
What We Learned	50
Making your own	51
Appendix: The Bundle's Keys	52
Appendix: Reader Versions	53
Acknowledgments	54

Preface

This is a book about reading in Emacs, and about how far that can be pushed before it stops being Emacs.

The reader it describes was built for one purpose: to read a poem in a language you are still learning, with the translations you trust beside it and a dictionary under every word, on whatever machine happens to be in your hand. The poem was Dante’s *Commedia*. The languages were Italian, English, and Russian. The machines ranged from a laptop running a graphical Emacs to an iPhone running Alpine Linux inside a terminal emulator, where every tap has to be translated into something a 1990s Info reader can understand.

What came out of that is a set of three Emacs Lisp files, a Python builder that writes Info files byte by byte, a package that verifies and installs its own updates, and a fork of the iSH terminal app — we called it 1Unix — whose whole reason for existing is to make Emacs’s Info mode behave like a touch application. None of this required patching Emacs. All of it required understanding Emacs rather better than we did when we started.

The book is written for the Emacs community in the plain sense: people who write Lisp, who have opinions about xterm-mouse-mode, who know what a tag table is and have perhaps wondered whether Info could carry a novel. It is organised as the reader was organised, from the file format up to the fingertip. Each chapter explains what we wanted, what Emacs already offered, what we had to add, and shows the Lisp — the actual code from the shipped reader, not a tidied illustration. Where we made mistakes that cost a week, the mistakes are here too, because they are the part of the story most likely to save you one.

Three of the chapters are the heart of the matter. *Touch* explains how a mode-line string becomes a button that works in a terminal, and why the obvious way of doing it fails on a phone. *Two Bars, One Contract* explains how the reader’s own bar and the terminal app’s native bar came to mirror each other, so that the native one can be turned off without losing a command. And *1Unix* explains what had to change on the other side of the terminal boundary: how a finger becomes a mouse report, why the reports had to be made atomic, and why a menu that chose “None” every time turned out to be a coordinate bug three layers below Emacs.

The reader is free software and the bundles for public-domain translations of Dante are published at First Pair Press. The Lisp quoted here is Reader 1.47, the version current as this book went to press.

Why Info

Ask most Emacs users where Info fits in their lives and they will say: it is where the manuals live. C-h i, a directory, the Elisp reference, n and p and u. It is old, it is plain, and it is not something anyone has reached for to read a *book* in a very long time.

We reached for it on purpose, and the reasons are worth stating, because they shaped everything that follows.

Info is hypertext with names. A node has a name; a cross-reference names a node in a manual; a manual has a directory entry; the whole thing is addressable as (manual)Node. Those are exactly the primitives a book with a graph underneath it needs: a chapter is a node, a citation is a cross-reference into a second manual of witnesses, a glossary is a node in that second manual, and the address of any of them can be written in a plain string. Info predates the web and has none of the web’s ambitions, which turns out to be an advantage — there is no layout engine to fight, only text, and text is what a poem is.

Info runs everywhere Emacs runs, including where nothing else does. The standalone info program reads the same files. Emacs in a terminal reads them. Emacs 27 on an emulated x86 Alpine inside an iPhone app reads them. A reader written as Info bytes plus Lisp has no build step on the reading machine, no dependencies, and no network. That constraint became a design principle: *everything works offline; no file in the bundle contacts a network.*

Info’s file format is simple enough to write by hand. This matters more than it sounds. The Info file is text with ^_ separators, a Tag Table of byte offsets, and a handful of conventions for menus and notes. A builder that writes those bytes itself, rather than through makeinfo, knows the exact line and column of every word it writes. The reader can then underline a word from a table of positions without searching, and can hide a translation by hiding a range of lines it was told about at build time. makeinfo is a fine program, but it does not tell you where your words went.

Info already has a reader with a keymap. n, p, u, l, SPC, RET, menus, g — the discipline of a reading mode is already there, and a minor mode can add to it without replacing it. Everything the First Pair reader does is layered over Info-mode through a minor mode, a few pieces of advice, and one hook. When the mode is off, the bundle is still a perfectly good pair of Info manuals.

There is a personal reason too, and it belongs in the record. In 1994, at the University of Pennsylvania, I wrote a hypertext in Texinfo, and I had in front of me Mark Liberman’s Spanish reader in Emacs — a text with a dictionary under it, thirty years before anyone said “language learning app”. Info was the natural home for a text that explains itself then, and nothing since has made it less so. This book acknowledges that debt.

What a parallel reader has to do

Before the machinery, the requirements, stated as a reader would state them.

1. Show the source text and one or more translations, unit by unit — for Dante, tercet by tercet — so the eye moves down one column of Italian and finds the English and Russian beside it, never a page away.
2. Let the reader choose *which* translations, per language, and change the choice while reading, without the source moving.

3. Explain any word of the source on demand: what it is, what it means, in the languages the reader has chosen, even when the poet spelled it his own way seven centuries ago.
4. Step through the source word by word, looking each up; and step through the *significant* words, skipping articles and prepositions and the verb *to be*.
5. Open the witnesses behind a quotation, or the notes behind a canto, in a second pane that never steals the reading position.
6. Remember where the reader stopped and what was on screen, and return there.
7. Do all of this by touch on a phone, in a terminal, with no keyboard on screen.
8. Update itself safely, and never load anything large before the first page.

Items one through six are what the Lisp does. Seven is what most of this book is about. Eight is where the builder and the package meet.

The Bundle on Disk

An Emacs bundle is a directory. Everything the reader will ever need is in it, and a manifest binds every file to a source commit. This is the layout as the delivery contract states it:

<code><bundle>/</code>	
<code>init.el</code>	loader: registers the bundle; uses an installed reader
<code>if present</code>	
<code>install.sh</code>	copies the manuals into an Info directory (install-
<code>info or Emacs)</code>	
<code>dir</code>	Info directory node listing both manuals
<code><stem>.info</code>	the book (over 300 KB: an Indirect main file
<code><stem>.info-1, -2, ...</code>	with its nodes in subfiles, as makeinfo splits)
<code><stem>-refs.info</code>	the references, split the same way when large
<code>texi/<stem>.texi</code>	Texinfo source of the book
<code>texi/<stem>-refs.texi</code>	Texinfo source of the references
<code>Guide.md, README.md</code>	the complete first-use manual (identical bytes)
<code>lisp/firstpair-bundle.el</code>	bundle loader and index access
<code>lisp/firstpair-lexicon.el</code>	offline dictionary
<code>lisp/firstpair-reader.el</code>	the reading mode
<code>data/bundle.json</code>	bundle description (schema firstpair-emacs-bundle-v1)
<code>data/reader.json</code>	ordered reader pages: id, title, node, part
<code>data/records.json</code>	records and evidence targets
<code>data/references.json</code>	every cross-reference each node contains, per manual
<code>data/marked.tsv</code>	words the dictionary window explains, by position
<code>data/regions.tsv</code>	aligned editions: the lines of each language in each
<code>unit</code>	
<code>data/regions.index.json</code>	byte offsets so a node's regions load alone
<code>data/translations.json</code>	the languages and editions of an aligned book
<code>lexicon/LEXICON.json</code>	lexicon description (schema firstpair-lexicon-v1)
<code>lexicon/entries.tsv</code>	id, headword, part, code, frequency, senses
<code>lexicon/forms/<letter>.tsv</code>	form, entry, features, enclitic – one shard per first
<code>letter</code>	
<code>lexicon/glosses/<letter>.tsv</code>	language, key, kind, headword, part, definitions,
<code>source</code>	
<code>lexicon/stems.tsv, endings.tsv</code>	for languages analysed by stem and ending
<code>FIRSTPAIR-EMACS-MANIFEST.json</code>	sealed inventory (schema firstpair-emacs-manifest-v1)

Two manuals, not one. The book is `<stem>.info`; everything the book points at — witnesses, records, evidence, the glossary — is `<stem>-refs.info`. The reader keeps them in separate windows, and that separation, decided on the first day, is the single design choice that most shaped the Lisp. A citation in the book is written as an ordinary Info cross-reference into the second manual:

```
(*note Atticus 4-8a: (cicero-on-himself-refs)Letters to Atticus 4-8a,)
```

Info hides the comma terminator, so the reader sees “(see Atticus 4-8a)”; the reader mode hides the manual name too. Following it opens the witness in the references window below, and the book does not move.

Node names

The first thing Info teaches anyone who writes it by hand is that node names are not free text. A node name may not contain a comma, a colon, a parenthesis, or a period — the period matters most, because Emacs truncates a labelled cross-reference at the first one. *Letters to Atticus 4.8a* is therefore the node

Letters to Atticus 4-8a, while its title and its citation label keep the period. The builder applies the rule everywhere; the reader never has to think about it. Chapter headings additionally carry Ref: anchors named <page id>-<heading slug>, so a link can land inside a node.

Why the builder writes the bytes

The Info file is written by `firstpair_emacs.infowriter`, a Python module that renders a document model straight to Info text and records, as it goes, the line and column of every word the lexicon can explain, and the line range of every language block in every unit. The same document model is rendered to Texinfo, which ships beside the Info file and is compiled by `makeinfo` during validation to prove the two agree. `makeinfo` could have produced the Info file; it could not have produced `marked.tsv`.

`marked.tsv` has one row per underlined word:

column	meaning
manual	manual stem
node	node name
line	1-based line within the node, counting the File: header line as 1
column	0-based character column
length	length of the word in characters
form	the normalised form the lexicon is keyed on
entries	comma-separated lexicon entry ids, best reading first

`regions.tsv` has one row per language block: manual, node, language (or translation id), unit, first line, last line, and whether the block is the source. For Dante that table has 56,362 rows; how the reader avoids parsing them all is the subject of the chapter on big books.

The manifest and the sealed inventory

`FIRSTPAIR-EMACS-MANIFEST.json` records the source commit, product, edition, counts, unmatched citation anchors, and the SHA-256 of every file. Validation begins by checking that inventory, and the publisher refuses a bundle whose commit is not the repository's HEAD. This sounds bureaucratic until the day a phone reports last week's reader while running this week's version; we will come to that.

Two Manuals, Three Windows

The reader's first job is to keep the book above and everything else below. Emacs has windows; what it lacks is a notion of what a window is *for*. The reader gives windows roles with a window parameter and never asks "which window is the references?" any other way:

```
(defun firstpair-reader--window (role)
  "Return the live window on the selected frame playing ROLE, or nil."
  (seq-find (lambda (window) (eq (window-parameter window 'firstpair-role) role))
    (window-list nil 'no-minibuffer)))

(defun firstpair-reader--claim (window role)
  "Record that WINDOW plays ROLE and return it."
  (set-window-parameter window 'firstpair-role role)
  window)
```

There are three roles: reader, references, and lexicon. A buffer's role, by contrast, is derived from the manual it shows:

```
(defun firstpair-reader--role (&optional buffer)
  "Return the role BUFFER plays: `reader', `references', or nil."
  (with-current-buffer (or buffer (current-buffer))
    (let ((bundle (firstpair-bundle-current)))
      (when bundle
        (let ((manual (firstpair-bundle-manual)))
          (cond ((equal manual (firstpair-bundle-reader bundle)) 'reader)
                ((equal manual (firstpair-bundle-reference bundle))
                 'references)))))))
```

`firstpair-bundle-current` deserves a word. It identifies the bundle by the *directory* of the Info file, not by the manual's name, because two editions of one book — a preview and the complete text, or the public Dante and the private study Dante — share manual names and must be registered together:

```
(defun firstpair-bundle-current ()
  "Return the bundle owning the Info manual in the current buffer, or nil.
The bundle is identified by the directory of the Info file, so several
editions of one book, whose manuals share a name, stay distinct."
  (when (and (derived-mode-p 'Info-mode) Info-current-file)
    (or (firstpair-bundle-for-file Info-current-file)
        (firstpair-bundle-for-manual
         (file-name-base (directory-file-name Info-current-file))))))
```

Registration itself is small. It loads `data/bundle.json`, adds the directory to `Info-directory-list` — after `info-initialize`, or every other manual on the machine disappears — and records the bundle:

```
(defun firstpair-bundle-register (root)
  "Register the bundle at ROOT and make its Info manuals findable."
  (let* ((bundle (firstpair-bundle-load root))
        (directory (directory-file-name (firstpair-bundle-root bundle))))
    ;; Let Info collect its default directories first; setting
    ;; `Info-directory-list' before that would hide every other manual.
    (info-initialize)
    (add-to-list 'Info-directory-list directory)
    (when (boundp 'Info-additional-directory-list)
```

```
(add-to-list 'Info-additional-directory-list directory))
(setf (alist-get directory firstpair-bundles nil nil #'equal) bundle)
bundle))
```

Redirecting a cross-reference

When the reader follows (cicero-on-himself-refs)Letters to Atticus 4-8a from the book, Info's own Info-goto-node would replace the book with the witness in the same window. The reader wants the witness *below*. The mechanism is advice on Info-goto-node, armed only while a reader command is running:

```
(defvar firstpair-reader--redirecting nil)

(defun firstpair-reader--redirect (nodename)
  "Return (ROLE . NODENAME) when NODENAME should open in another window, else nil.
The returned NODENAME addresses the manual by file inside its bundle."
  (let* ((target (firstpair-reader--parse-target nodename))
         (bundle (and target (firstpair-reader--target-bundle (car target)))))
    (when bundle
      (let ((role (if (equal (car target) (firstpair-bundle-reference bundle))
                      'references
                      'reader)))
        (unless (eq role (firstpair-reader--role))
          (cons role (firstpair-reader--node bundle (car target) (cdr target)))))))

(defun firstpair-reader--goto-node-advice (original nodename &rest arguments)
  "Open NODENAME in the window its manual belongs to while a reader command runs.
ORIGINAL is `Info-goto-node'; ARGUMENTS are passed through to it."
  (let ((redirect (and firstpair-reader--redirecting
                       (firstpair-reader--redirect nodename))))
    (if redirect
        (firstpair-reader--goto (car redirect) (cdr redirect))
        (apply original nodename arguments))))

(defun firstpair-reader-follow-nearest-node (&optional fork)
  "Follow the reference or menu item at point.
References into the book's other manual open in that manual's window."
  (interactive "P")
  (let ((firstpair-reader--redirecting t))
    (Info-follow-nearest-node fork)))
```

The minor mode binds [remap Info-follow-nearest-node], [remap Info-mouse-follow-nearest-node], and [remap Info-follow-reference] to the redirecting versions. Outside those commands — g, l, a menu — Info behaves exactly as it always has. The advice is dynamic-scope Lisp at its most useful: one let around one call, and no global state survives it.

Making room

The references window is created below the reader at a fraction of its height; the dictionary is created below the references, or below the reader when there are no references. On a laptop that is all there is to it. On a phone in portrait, forty rows tall with a menu bar, a header row, two mode lines, and a keyboard that may appear at any time, split-window fails as often as it succeeds, and the reader had to learn to *borrow*:

```
(defun firstpair-reader--borrow-window (window role)
  "Temporarily assign WINDOW to ROLE, remembering what it displayed."
  (set-window-parameter window 'firstpair-borrowed-role
    (window-parameter window 'firstpair-role))
  (set-window-parameter window 'firstpair-borrowed-buffer (window-buffer window))
  (set-window-parameter window 'firstpair-borrowed-start (window-start window))
  (set-window-parameter window 'firstpair-borrowed-point (window-point window))
  (set-window-parameter window 'firstpair-borrowed-hscroll (window-hscroll window))
  (firstpair-reader--claim window role))
```

When the references pane is idle at its Top node, the dictionary simply takes it over; when the dictionary closes, `firstpair-reader--restore-borrowed-window` puts the references back with their scroll position intact. The decision logic lives in `firstpair-reader--ensure-window`, whose lexicon branch reads, in order: replace an idle references pane; else split the references; else split the reader; else borrow the references anyway; else give up with a user error. Each of those branches was added by a phone that had failed the previous one.

One detail from that function repays attention. The dictionary window is split off at `window-min-height` and only *then* grown to fit its content:

```
(defun firstpair-reader--fit-lexicon-window (&optional window)
  "Fit the dictionary WINDOW to its body, up to its configured maximum."
  (let ((target (or window (firstpair-reader--window 'lexicon))))
    (when (and (window-live-p target)
      (eq (window-parameter target 'firstpair-role) 'lexicon))
      (condition-case nil
        (progn
          (with-current-buffer (window-buffer target)
            (set-window-point target (point-min))
            (set-window-start target (point-min) t))
          (fit-window-to-buffer target
            (max firstpair-reader-lexicon-height window-safe-
              min-height)
            window-safe-min-height))
        (error nil))))))
```

Asking for the full ten lines up front made an otherwise viable split fail on the phone; asking for the minimum and fitting afterwards succeeds, and a compact entry of a headword and four senses takes five rows instead of ten. Small windows are a resource, and the reader treats them as one.

Every Word Answers

The dictionary is the reason the reader exists. A learner reading Dante meets *ch'i'* and *eterno* and *avea* in the first canto and needs, each time, an answer that says what the word is — “apocope of *avere*, third singular imperfect” — and what it means, in the languages she reads, without leaving the tercet. This chapter is how a word on the screen becomes that answer.

Marked words and where to find them

For prose editions — the Cicero, where the Latin quotations sit inside English — the builder writes `marked.tsv`: for every word the lexicon can explain, its node, line, column, and length. On entering a node the reader underlines them:

```
(defun firstpair-reader--mark (bundle)
  "Underline the words of the current node that BUNDLE's lexicon explains."
  (firstpair-reader--unmark)
  (when firstpair-reader-highlight
    (dolist (entry (firstpair-bundle-marked-words bundle (firstpair-bundle-manual)
Info-current-node))
      (let ((region (firstpair-reader--locate entry)))
        (when region
          (let ((overlay (make-overlay (car region) (cdr region))))
            (overlay-put overlay 'firstpair-marked entry)
            (overlay-put overlay 'firstpair-bundle bundle)
            (overlay-put overlay 'face 'firstpair-reader-marked)
            (overlay-put overlay 'help-echo #'firstpair-reader--help-echo)
            (overlay-put overlay 'evaporate t)
            (push overlay firstpair-reader--overlays))))))
    (setq firstpair-reader--overlays
      (sort firstpair-reader--overlays
        (lambda (a b) (< (overlay-start a) (overlay-start b)))))))
```

Overlays, not text properties, because the buffer is Info's and must stay pristine; evaporate so an overlay that loses its text disappears rather than lingering as a zero-width ghost. The interesting function is `--locate`. The builder's line and column are exact for the bytes it wrote, but Info's own rendering may have hidden a *note or an anchor on the same line; so the reader checks that the text at the recorded position normalises to the recorded form, and if not, searches the same line for a word that does:

```
(defun firstpair-reader--locate (entry)
  "Return (START . END) of the marked word ENTRY in the current node, or nil.
The builder records the line and column of every marked word; when Info's
rendering has moved the text, the word is found again on the same line."
  (save-excursion
    (goto-char (point-min))
    (when (zerop (forward-line (1- (plist-get entry :line))))
      (let* ((start (+ (point) (plist-get entry :column)))
              (end (+ start (plist-get entry :length)))
              (form (plist-get entry :form))
              (bundle (firstpair-bundle-current))
              (limit (line-end-position)))
        (if (and (<= end limit)
                  (equal (firstpair-lexicon-normalise (buffer-substring-no-properties
```

```

start end) bundle)
      form))
    (cons start end)
  (let (found)
    (while (and (not found) (re-search-forward "[[:alpha:]]+" limit t))
      (when (equal (firstpair-lexicon-normalise (match-string-no-properties
0) bundle) form)
        (setq found (cons (match-beginning 0) (match-end 0))))))
    found))))))

```

Two decisions are hidden in that loop. The exact-position check is the fast path, and it is the path the validator counts: `firstpair-check` opens every node and insists that the number of overlays equals the number of rows in `marked.tsv`, so a builder that miscounts a column is caught at validation, not by a reader. And the fallback is deliberately limited to the same line: a word that has wandered to another line means the builder and the file disagree, which is an error, not a case to be papered over.

Aligned editions: every source word

Dante is different. In an aligned edition every Italian word is a candidate, and underlining twelve thousand words would turn the poem into a rug. So aligned editions carry no underlines at all; `marked.tsv` for the study Dante has seven rows. Instead the reader knows, from `regions.tsv`, which lines are source lines, and any word in them is looked up on demand:

```

(defun firstpair-reader--source-spans (bundle)
  "Buffer spans (START . END) of the source-language regions of the current node."
  (let (spans)
    (dolist (region (firstpair-reader--regions bundle) (nreverse spans))
      (when (plist-get region :source)
        (save-excursion
          (goto-char (point-min))
          (when (zerop (forward-line (1- (plist-get region :start))))
            (let ((start (point)))
              (forward-line (1+ (- (plist-get region :end) (plist-get
region :start)))))
              (push (cons start (point)) spans)))))))

(defun firstpair-reader--source-word-at-point (bundle)
  "Return the source-language word at point in BUNDLE, or nil."
  (let ((bounds (bounds-of-thing-at-point 'word)))
    (when (and bounds
                (firstpair-reader--in-spans-p
                 (car bounds) (firstpair-reader--source-spans bundle)))
      (buffer-substring-no-properties (car bounds) (cdr bounds)))))

```

A tap on a word in an Italian line passes the surface form straight to the lexicon. A tap on an English line does nothing — it is not a query — which is exactly what a reader expects and what took a while to get right, because the first version happily looked up “forest” in an Italian dictionary and reported failure.

Walking the source

The word arrows are the most used controls in the reader, and they walk the Italian, never the translations. --move-marked does the right thing for both kinds of edition — in prose it steps between marked overlays, in an aligned text between words inside the source spans:

```
(defun firstpair-reader--move-source-word (forward spans)
  "Move to the next word inside SPANS, or the previous unless FORWARD.
  Return non-nil when a word was found."
  (let ((origin (point)) (found nil))
    (save-excursion
      (catch 'done
        (while (if forward (< (point) (point-max)) (> (point) (point-min)))
          (if forward (forward-word 1) (backward-word 1))
          (let ((start (if forward (save-excursion (backward-word 1) (point))
                                (point))))
            (when (and (/= start origin)
                      (firstpair-reader--in-spans-p start spans)
                      (if forward (> start origin) (< start origin)))
              (setq found start) (throw 'done t))))))
    (when found (goto-char found) t)))
```

The *significant* arrows — << and >>, J and K — apply one more test. Reading Dante word by word means stopping at *e* and *di* and *che* and *era* far more often than at anything worth looking up. A stop-word list, a defcustom so another source language can replace it, is skipped:

```
(defcustom firstpair-reader-significant-stopwords
  '("a" "ad" "al" "alla" "alle" "allo" "ai" "agli"
    "da" "dal" "dalla" "dalle" "dallo" "dai" "dagli"
    "di" "del" "della" "delle" "dello" "dei" "degli"
    "in" "nel" "nella" "nelle" "nello" "nei" "negli"
    "con" "su" "sul" "sulla" "sulle" "sullo" "sui" "sugli"
    "per" "tra" "fra" "e" "ed" "o" "od" "ma" "che" "se"
    "il" "lo" "la" "i" "gli" "le" "un" "uno" "una"
    "mi" "ti" "si" "ci" "vi" "ne" "io" "tu" "egli" "ella" "noi" "voi"
    "essere" "esser" "sono" "sei" "è" "siamo" "siete" "era" "eri" "erano"
    "fui" "fosti" "fu" "fur" "furono" "sia" "siano" "fosse" "fossero"
    "sarà" "saranno" "sarei" "sarebbe" "avere" "aver" "ho" "hai" "ha"
    "hanno" "aveva" "avevano" "ebbe" "ebbero" "abbia" "abbiano")
  "Lowercase source words skipped by significant-word movement."
  :type '(repeat string))

(defun firstpair-reader--significant-word-p ()
  "Return non-nil when the source word at point carries lexical weight."
  (let ((word (downcase (or (thing-at-point 'word t) ""))))
    (and (not (string-empty-p word))
         (not (member word firstpair-reader-significant-stopwords)))))

(defun firstpair-reader--move-significant (forward)
  "Move to the next significant source word, or previous unless FORWARD."
  (let ((origin (point)) found)
    (condition-case nil
      (while (not found)
```

```

      (firstpair-reader--move-marked forward)
      (when (firstpair-reader--significant-word-p)
        (setq found t)))
    (user-error
      (goto-char origin)
      (user-error (if forward "No further significant words"
        "No earlier significant words"))))
  found))

```

The condition-case around the loop is there because --move-marked signals a user-error at the end of the canto; the significant mover catches it, returns point to where the reader was, and re-signals with its own message. Without the goto-char, a reader who pressed >> at the last noun of a canto found point stranded on the final *e*.

The current word stays underlined

When a lookup happens, the word looked up gets its own overlay, at a priority above the marks, and it stays while the dictionary is open — so the eye can come back from the definition to the line and find its place:

```

(defun firstpair-reader--underline-current-word (bundle word)
  ...) ; one overlay, `firstpair-reader-current-word' face, replaced on every
lookup

```

Every word arrow moves that underline. Closing the dictionary removes it. This is a small thing that turned out to be the difference between a dictionary you consult and a dictionary you read *with*; on a phone, where the dictionary pane is four rows below the poem, it is the thread back.

The lexicon: normalise, analyse, gloss

Everything the dictionary knows is in tab-separated tables, loaded lazily into hash tables and keyed on a *normalised* form. Normalisation is a rule the bundle declares — for Italian: lower case, keep the diaereses of the Gutenberg text as plain vowels, strip apostrophes — and the reader applies exactly that rule to whatever it finds in the buffer:

```

(defun firstpair-lexicon-normalise (word &optional bundle)
  "Fold WORD to the shape the delivered tables are keyed on.
The rule is BUNDLE's, or the rule of `firstpair-lexicon--spec-bundle', or
Latin's: lower case, combining marks stripped or kept, replacements applied."
  (let* ((spec (firstpair-lexicon-spec (or bundle firstpair-lexicon--spec-bundle)))
    (combining (or (alist-get 'combining spec) "strip")))
    (text (or word ""))
    (text (if (equal combining "strip")
      (let ((plain (ucs-normalize-NFD-string text)))
        (apply #'string
          (seq-filter (lambda (character)
            (not (memq (get-char-code-property
              character 'general-category)
                '(Mn Mc Me))))
            (append plain nil))))
      (ucs-normalize-NFC-string text)))
    (text (if (equal combining "diaeresis")
      (apply #'string (mapcar (lambda (character)
        (or (alist-get character firstpair-

```

```
lexicon-diaereses) character))
                                (append text nil)))
      text))
      (text (if (alist-get 'lowercase spec) (downcase text) text))
      (text (string-trim text (format "[%s]+" (or (alist-get 'strip spec) "'"))
                                (format "[%s]+" (or (alist-get 'strip spec) "'"))))))
      (dolist (pair (alist-get 'replace spec))
        (setq text (replace-regexp-in-string (regexp-quote (car pair)) (cadr pair) text
t t)))
      (replace-regexp-in-string "[^[:alpha:]]'" "" text)))
```

Analysis then tries, in order: the form itself in the forms table; the form with an enclitic pronoun removed (Dante’s *dirmi*, *vedermi* — the bundle lists its enclitics, longest first); and, for a language analysed by stem and ending like Latin, the stem-and-ending fallback:

```
(defun firstpair-lexicon-analyse (bundle word)
  "Return the readings of WORD offered by BUNDLE, best first."
  (let* ((form (firstpair-lexicon-normalise word bundle))
        (direct (and (> (length form) 0) (gethash form (firstpair-lexicon--forms-
table bundle form)))))
    (or direct
      (seq-some (lambda (enclitic)
                  (and (> (length form) (1+ (length enclitic)))
                      (string-suffix-p enclitic form)
                      (let ((base (substring form 0 (- (length form) (length
enclitic)))))
                        (mapcar (lambda (reading)
                                  (plist-put (copy-sequence reading) :enclitic
enclitic))
                              (gethash base (firstpair-lexicon--forms-table
bundle base))))))
                (firstpair-lexicon-enclitics bundle))
        (firstpair-lexicon--fallback bundle form)))))
```

The hard part of Italian is not in the Lisp at all; it is in the builder. The English Wiktionary extraction published by Kaikki lists every lemma with its senses and inflected forms, but Dante’s spelling is his own. The Python analyser tries the exact form and then a short, ordered list of *restorations* — elided articles and clitics (*l'*, *ch'*), truncated words (*amor*, *cammin*, *fuor*), old verb endings (*dicea*, *avea*), diaereses (*sapienza*) — and records which restoration succeeded, so the reader can say “apocope of *amore*” instead of guessing. The forms table the bundle ships is the *result* of that analysis, projected onto the words the poem actually uses: 11,923 forms over 10,338 entries for the study Dante. The reader never has to be clever about Italian; the builder was clever once.

The compact dictionary

The dictionary pane on a phone has four or five rows. Reader 1.17 redesigned it for that: one bold line of Italian headwords (all the lemmas the readings point at, --separated, or the surface word itself when analysis found nothing), then up to two sense lines per selected language, unwrapped, with **m** — or the **More** button — revealing the rest:

```
(defun firstpair-lexicon--insert-definitions (bundle word readings)
  "Insert compact sense lines for WORD from BUNDLE in each selected language.
Return non-nil when at least one sense was inserted."
```

```

(setq firstpair-lexicon-has-more nil)
(let (inserted)
  (dolist (language (firstpair-lexicon-selected bundle))
    (let* ((definitions (firstpair-lexicon-definitions
                        bundle (alist-get 'id language) word readings))
           (senses (firstpair-lexicon--sense-lines definitions)))
      (when (> (length senses) 2)
        (setq firstpair-lexicon-has-more t)
        (dolist (sense (if firstpair-lexicon-expanded senses (seq-take senses 2)))
          (setq inserted t)
          (insert sense "\n"))))
    inserted))

```

Which senses come first is decided by a ranking that prefers glosses whose headword is written exactly as the entry's, then those with the same part of speech, and always admits glosses reached “via” another language — that is the Russian-through-English pivot, described in *The Graph Book*, by which an Italian word with no Russian entry is glossed through the English words of its own senses. A form no dictionary explains is reported as such: “No entry in the selected dictionaries.” Absence is stated, never filled by machine translation.

The dictionary answers in whichever languages the reader has selected — t cycles English, Russian, both — and the same selection governs which translations are visible. That coupling was deliberate: the languages you read in are the languages you look words up in.

Translations as State

Nothing in the reader changed as many times as the way translations are chosen, and the changes trace a lesson worth the whole chapter: *a control that cycles hides state; a control that toggles shows it.*

Where it started

The first Dante had two languages and one translation each. C-c C-t cycled English, Russian, both; the reader hid the other language's lines with invisible overlays. Then the public-domain hunt found four English and six Russian translations, and the question became which one. Version one: C-c C-v *rotated* the translation under point through the language's editions. Version two added a *second slot*: C-c C-b showed one more edition of the same language under the first, and rotated it independently. Every command was a cycle; the only way to know what you would get was to press and see.

On a laptop that was tolerable. On a phone, where each press is a tap on a two-character button and the echo line is the only feedback, it was maddening. Which button rotated which slot depended on where point happened to be. Reaching Cary when Longfellow and Sibbald were showing took three or four taps and a mental model of a state machine that the screen did not display.

The model that stuck

Reader 1.39 replaced the slots with two pieces of state, both small enough to print:

```
(defvar firstpair-reader-translation-selections nil
  "Alist of language id to the ordered list of translation ids it shows.
  A language without an entry shows its default translation.")
```

```
(defvar firstpair-reader-language-order nil
  "Translation language ids in the order their blocks appear on screen.
  Languages not listed follow, in the edition's declared order.")
```

Everything else is derived. The editions actually on screen for a language are the selected ones that cover the current part — the *Inferno* has translations the *Paradiso* lacks — with a fallback to the language's default so a page never goes blank:

```
(defun firstpair-reader--effective-translations (bundle lang)
  "Return the ordered translation ids LANG shows on the current page.
  Selected ids that do not cover this part are skipped; when none remain,
  the language's default (or first) covering edition fills in, so a page
  never goes blank."
  (let* ((items (firstpair-reader--candidates bundle lang))
        (ids (mapcar (lambda (item) (alist-get 'id item)) items))
        (chosen (alist-get lang firstpair-reader-translation-selections nil nil
                           #'equal)))
    (effective (seq-filter (lambda (id) (member id ids)) chosen)))
  (or effective
    (let ((pick (or (seq-find (lambda (item) (eq (alist-get 'default item) t))
                           items)
                    (car items))))
      (and pick (list (alist-get 'id pick)))))))
```

Hiding is still overlays. The regions of the current node whose translation is not among the shown ids become invisible, each with the blank line that follows it, so the tercet closes up:

```
(defun firstpair-reader--apply-regions (bundle)
  "Hide the translation regions of the current node that are not shown."
  (let ((chosen (if (firstpair-bundle-translations bundle)
                    (firstpair-reader--shown-translations bundle)
                    (mapcar (lambda (item) (alist-get 'id item)) (firstpair-lexicon-
selected bundle))))))
    (dolist (region (firstpair-reader--regions bundle))
      (unless (or (plist-get region :source) (member (plist-get region :language)
chosen))
        (save-excursion
          (goto-char (point-min))
          (when (zerop (forward-line (1- (plist-get region :start))))
            (let ((start (point)))
              (forward-line (1+ (- (plist-get region :end) (plist-get
region :start))))
              ;; Swallow the blank line that separates this region from the next.
              (when (and (not (eobp)) (looking-at-p "^$")) (forward-line 1))
              (let ((overlay (make-overlay start (point))))
                (overlay-put overlay 'invisible t)
                (overlay-put overlay 'firstpair-region region)
                (overlay-put overlay 'evaporate t)
                (push overlay firstpair-reader--overlays))))))))))
```

The source is never hidden. That rule — the Italian never moves — is what makes it possible to change translations mid-tercet without losing the line.

Checkbox menus

With a list per language, the menu becomes obvious. In a terminal the menu bar carries **Tr-Eng** and **Tr-Rus**; each lists **None** and then every edition covering the current part, and every shown edition is checked. Select an unchecked one and it is added beside the others; select a checked one and it is removed; remove the last and the language hides. The menu is built by a `:filter` so it is always current:

```
(define-key-after firstpair-reader-mode-map [menu-bar translation-english]
  '(menu-item "Tr-Eng" ignore
    :filter firstpair-reader--terminal-english-menu
    :visible (and (not (display-graphic-p))
      (firstpair-reader--translation-language-available-p
"en")))
    :help "Choose or hide the English translation")
  'translations)
```

```
(defun firstpair-reader--translation-language-menu (lang)
  "Build the terminal translation submenu for language LANG.
Every shown edition carries a checked box; selecting an unchecked one
shows it as well, selecting a checked one hides it."
  (let* ((bundle (firstpair-reader--bundle))
    (language (seq-find (lambda (item) (equal (alist-get 'id item) lang))
      (firstpair-bundle-translation-languages bundle)))
    (visible (firstpair-reader--translation-language-visible-p bundle lang))
    (effective (and visible (firstpair-reader--effective-translations bundle
lang))))
```

```

(none-command
 (firstpair-reader--translation-language-menu-command lang nil))
(entries (list (list none-command 'menu-item "None" none-command
                  :help "Hide this translation language"
                  :button (cons 'radio (not visible))))))
(dolist (item (firstpair-reader--candidates bundle lang))
 (let* ((id (alist-get 'id item))
        (command
         (firstpair-reader--translation-language-menu-command lang id)))
 (setq entries
  (append entries
   (list
    (list command 'menu-item
          (firstpair-reader--translation-title bundle id)
          command
          :help "Show or hide this edition"
          :button (cons 'checkbox
                        (and (member id effective) t)))))))
 (append (list 'keymap
              (format "%s translations"
                      (or (alist-get 'label language) (upcase lang))))
  entries)))

```

Notice `firstpair-reader--translation-language-menu-command`. It interns a name like `firstpair-reader-menu-en-en-cary` and fsets it to a small interactive lambda. The first version used anonymous closures, and stock Emacs tmm was perfectly happy with them — but the terminal host's menu bridge was not, and every item ran the first command. A menu item that a terminal app can serialise must be a *named* command. This is one of several places where the terminal taught the Lisp its manners.

The toggle itself:

```

(defun firstpair-reader-toggle-language-translation (lang id)
  "Show translation ID of LANG when hidden; hide it when on screen.
Hiding the last shown edition hides the language; ID nil hides the
language outright."
  (let* ((bundle (firstpair-reader--bundle))
         (visible (firstpair-reader--translation-language-visible-p bundle lang))
         (effective (and visible (firstpair-reader--effective-translations bundle
lang))))
    (cond
     ((null id)
      (firstpair-reader--hide-translation-language bundle lang))
     ((member id effective)
      (let ((rest (remove id effective)))
        (if rest
         (setf (alist-get lang firstpair-reader-translation-selections nil nil
#'equal)
              rest)
         (firstpair-reader--hide-translation-language bundle lang))))
     (t
      (unless (seq-find (lambda (item) (equal (alist-get 'id item) id))

```

```

      (firstpair-reader--candidates bundle lang))
    (user-error "That translation does not cover this part"))
  (unless visible (firstpair-reader--show-translation-language bundle lang))
  (setf (alist-get lang firstpair-reader-translation-selections nil nil #'equal)
        (append effective (list id))))
  (firstpair-reader--after-translation-change bundle lang)))

```

Feedback that survives the menu

A TTY menu erases the echo area when it closes — *after* the command has run. So the reader’s confirmation (“English: Longfellow (1867) + Cary (1814) ≈”) vanished before anyone saw it. The fix is to post the message from a one-shot post-command-hook, which runs after the menu’s own teardown:

```

(defun firstpair-reader--schedule-terminal-language-feedback (bundle lang &optional
  _id)
  "Keep LANG's shown edition list visible after a TTY menu tap."
  (let* ((language (seq-find (lambda (row) (equal (alist-get 'id row) lang))
                             (firstpair-bundle-translation-languages bundle)))
        (ids (and (firstpair-reader--translation-language-visible-p bundle lang)
                  (firstpair-reader--effective-translations bundle lang)))
        (label (format "%s: %s"
                       (or (alist-get 'label language) (upcase lang))
                       (if ids
                           (mapconcat (lambda (id)
                                         (firstpair-reader--translation-title bundle
                                           id))
                                       ids " + ")
                           "None"))))
    ;; A terminal menu invokes this command before its own teardown. Posting
    ;; from the outer command loop leaves the result in the echo area after
    ;; that teardown instead of letting the menu erase it.
    (setq firstpair-reader--terminal-translation-feedback label)
    (remove-hook 'post-command-hook
                  #'firstpair-reader--show-terminal-translation-feedback)
    (add-hook 'post-command-hook
              #'firstpair-reader--show-terminal-translation-feedback)))

```

The test for this checks that the hook is present after the command, fires it, captures the message, and then checks that the hook has removed itself. One-shot hooks that forget to remove themselves are how Emacs sessions slowly fill with ghosts.

The header row: tap to hide, tap to reorder

The row directly under the menu bar is the header line of the book’s window, and it shows what the state variables hold: EN Longfellow ◀Cary | RU Мин. It is not a report. Every element is a button:

- the edition’s **name** hides it;
- the ◀ before an edition moves it one step earlier within its language;
- the language tag (**EN**, **RU**) brings that language’s whole block first.

The row is rebuilt as a mode-line format list on every redisplay; the buttons are the same `firstpair-reader--button` strings the bars use, and their commands are, again, interned names so a terminal can dispatch them:

```
(defun firstpair-reader--header-command (action lang id)
  "Return a stable named command for header-row ACTION on LANG and ID."
  (let ((command (intern (format "firstpair-reader-header-%s-%s-%s"
                                action lang (or id "none")))))
    (fset command
      `(lambda ()
         ,(format "Header row control: %s %s %s." action lang (or id ""))
         (interactive)
         (cond ((eq ',action 'first) (firstpair-reader-language-first ,lang))
               ((eq ',action 'earlier)
                (firstpair-reader-move-translation-earlier ,lang ,id))
               (t (firstpair-reader-toggle-language-translation ,lang ,id))))
      command)))
```

Reordering has a consequence that the chapter on big books takes up: the order of translation blocks is baked into the Info text at build time. Hiding is an overlay; reordering means the reader must physically move lines. It does, unit by unit, under invariants that keep the Italian in place.

Tr< and Tr>: swap, not cycle

The rotate command survived, redefined: Tr< and Tr> (and [,], and a long press or right click on a line) swap the edition *under point* for the previous or next edition of its language that is not already on screen, keeping its position in the list. The reader stays in the block that changed:

```
(defun firstpair-reader--replace-translation (bundle lang id step)
  "Swap edition ID of LANG for its next unshown alternative, by STEP.
The replacement takes ID's place in the language's display order, and
point stays in the block that changed."
  ...
  ;; Keep point in the block that changed. Otherwise the old region
  ;; becomes invisible and a second tap can accidentally target another.
  (when (and origin (not (plist-get origin :source)))
    (let ((replacement
          (seq-find
            (lambda (region)
              (and (equal (plist-get region :language) next)
                   (equal (plist-get region :unit) (plist-get origin :unit))))
            (firstpair-reader--regions bundle))))
      (when replacement
        (goto-char (point-min))
        (forward-line
          (+ (1- (plist-get replacement :start))
             (min (or origin-offset 0)
                  (- (plist-get replacement :end)
                     (plist-get replacement :start))))))
      next)))
```

The comment records a bug that took an afternoon: after a swap the old region became invisible, point stayed at its buffer position, which now lay inside the *next* visible block, and the next Tr> swapped the wrong translation.

2nd: the favorites button

One control did not fit the toggle model and earned its own semantics. A reader who studies with Michael Palma's English and Alexander Ilyushin's Russian, but reads the public-domain Longfellow and Min beside them for contrast, wants a single tap that brings the favourites back — without disturbing anything else on screen — and a single hold that strips the screen back down to just them. That is 2nd, and b and B:

```
(defcustom firstpair-reader-favorite-translations
  '(("en" . "en-palma")
    ("ru" . "ru-ilyushin"))
  "Preferred comparison translation for each language.
The `firstpair-reader-second-translation' command adds every configured
favorite available in the current bundle and part. Its inverse,
`firstpair-reader-only-favorite-translations', keeps those favorites and
removes the other editions of the same languages. Translation ids absent
from a bundle are ignored."
  :type '(alist :key-type (string :tag "Language id")
                :value-type (string :tag "Translation id"))
  :group 'firstpair)

(defun firstpair-reader-second-translation ()
  "Add the configured favorite translations to the editions on screen.
This is the ordinary `b' or 2nd action. It never removes an edition;
use `firstpair-reader-only-favorite-translations' to keep only favorites."
  (interactive)
  (let* ((bundle (firstpair-reader--bundle))
        (favorites (firstpair-reader--available-favorite-translations bundle))
        (changed)
        (unless favorites
          (user-error "This bundle has none of the configured favorite translations"))
        (dolist (pair favorites)
          (let* ((lang (car pair))
                 (id (cdr pair))
                 (visible (firstpair-reader--translation-language-visible-p bundle lang))
                 (effective (and visible (firstpair-reader--effective-translations bundle
lang))))
            (unless (member id effective)
              (unless visible (firstpair-reader--show-translation-language bundle lang))
              (setf (alist-get lang firstpair-reader-translation-selections nil nil
#'equal)
                    (append effective (list id)))
              (setq changed t))))
        (if changed
          (firstpair-reader--after-favorite-change bundle "Added favorites")
          (message "Favorite translations are already shown: %s"
                  (firstpair-reader-translations-label bundle))))))
```

The name `second-translation` is a fossil from the slot era, kept so `C-c C-b` and the strip's `b` key mean what they always meant to the fingers. The behaviour is additive and idempotent: pressing 2nd twice does nothing the second time and says so. Its inverse, `firstpair-reader-only-favorite-`

translations, is the long-press action on the same button, and the two together are the fastest route between “everything I compare” and “just what I study”.

What is left of the old commands

C-c C-t and t still cycle languages, because a three-state cycle is short enough to be legible, and T cycles it backward. = prints the current translations without changing anything — a reader on a phone asked for a way to *look* before touching — and the graphical **Translations** menu’s first item is that same report, live. The persisted state is exactly the two variables above plus the language selection, and the resume chapter shows how old saved states with :choices and :seconds are migrated so nobody’s bookmark broke when the slots went away.

Touch

Everything in this chapter exists because of one sentence in a message from a phone: “*ish emacs does not react to touch. can I configure it to bring the pointer to the touch?*”

The answer turned out to be yes, and then no, and then yes with conditions, and the conditions are the architecture. What follows is the Emacs half of it — how a mode-line string becomes a button, how a tap reaches it in a terminal, and the six or seven ways that went wrong before it went right. The next chapter is the other half, on the far side of the terminal.

What a terminal tells Emacs about a finger

A terminal emulator with mouse reporting on sends escape sequences for presses, releases, and (in some modes) movement: ESC [< 0 ; 12 ; 3 M for a press at column 12, row 3. Emacs decodes these when xterm-mouse-mode is on and produces the same mouse-1, down-mouse-1, drag-mouse-1 events a graphical frame would, with posn structures that say which window, which area (mode-line, header-line, or the text), and which string and offset inside a mode-line format the click landed on. The Lisp that handles a click can therefore be the same for a terminal and a GUI — *if* the terminal reports faithfully. Emacs itself never turned the mode on for us; the reader does, and on a terminal it also has to deal with what iSH sends when mouse reporting is *off*:

```
(defun firstpair-reader--apply-touch (bundle)
  "Give the current reader buffer its button bar and turn on mouse reporting.
  Terminals that report only focus (iSH sends ESC [ I on a tap) have those
  sequences decoded as focus events, so a tap never reads as M-[ I."
  (when firstpair-reader-touch
    ...
    (unless (display-graphic-p)
      (unless (bound-and-true-p xterm-mouse-mode)
        (ignore-errors (xterm-mouse-mode 1)))
      ;; A terminal that reports focus (iSH does, on every tap) sends ESC [ I / ESC
[ 0;
      ;; decode them to events and let them fall through silently.
      (define-key input-decode-map "\e[I" [focus-in])
      (define-key input-decode-map "\e[O" [focus-out])
      (dolist (event '([focus-in] [focus-out]))
        (unless (lookup-key global-map event)
          (define-key global-map event #'ignore))))))
```

The focus sequence was the first surprise. The user, reading with a terminal key logger, reported “a tap is: ESC [I”. iSH sends the focus-in sequence on every tap whether or not anything has focus, and Emacs 27 in a terminal, lacking a decoding for it, read it as M-[I — an undefined key, with a beep and an error in the echo area on every touch. Decoding it to [focus-in] and binding that to ignore made taps silent; turning on xterm-mouse-mode made them *events*.

A button is a propertized string

Emacs’s mode line and header line are format lists, and a string in one may carry a local-map text property. A click on the string looks up the event in that map. That is the whole mechanism of a button; the reader’s version is careful about which events it claims:

```
(defun firstpair-reader--button (label command &optional help width)
  "A bar button LABEL running COMMAND on a tap, with HELP as tooltip."
```

When `WIDTH` is non-nil, pad the button to that many terminal columns.
 The command runs with the tapped window selected, so a button on the book
 acts on the book even while the dictionary window has focus."

```
(let* ((map (make-sparse-keymap))
      (action (lambda (event)
                  (interactive "e")
                  (firstpair-reader--run-button-command event command)))
      (label-width (string-width label))
      (button-width (max (+ label-width 2) (or width 0)))
      (padding (- button-width label-width))
      (left-padding (/ padding 2))
      (right-padding (- padding left-padding)))
  (define-key map [header-line mouse-1] action)
  (define-key map [header-line drag-mouse-1] action)
  (define-key map [header-line down-mouse-1] #'ignore)
  (define-key map [header-line double-mouse-1] #'ignore)
  (define-key map [header-line triple-mouse-1] #'ignore)
  (define-key map [header-line mouse-2] action)
  (define-key map [mode-line mouse-1] action)
  (define-key map [mode-line drag-mouse-1] action)
  ;; Own the press as well as the release. Otherwise the standard mode-line
  ;; map begins a resize before our release handler runs; a retry can then be
  ;; interpreted as a double-click and maximize or replace a reading pane.
  (define-key map [mode-line down-mouse-1] #'ignore)
  (define-key map [mode-line double-mouse-1] #'ignore)
  (define-key map [mode-line triple-mouse-1] #'ignore)
  (define-key map [mode-line mouse-2] action)
  (define-key map [mouse-1] action)
  (define-key map [drag-mouse-1] action)
  (define-key map [down-mouse-1] #'ignore)
  (define-key map [double-mouse-1] #'ignore)
  (define-key map [triple-mouse-1] #'ignore)
  (firstpair-reader--bind-touch-scroll map)
  (proptertize (concat (make-string left-padding ?\s)
                      label
                      (make-string right-padding ?\s))
               'face '(:box (:line-width 1) :inherit mode-line-highlight)
               'mouse-face 'highlight 'local-map map
               'firstpair-reader-command command
               'help-echo (if (display-graphic-p)
                              (or help label)
                              #'firstpair-reader--no-help-echo))))
```

Each of those `define-key` lines is a scar. Read them as a list of things a finger does that a mouse does not:

- **A finger drifts.** Between touch-down and touch-up it moves a few pixels, and on a 7-pixel-wide terminal cell that is a different column. Emacs then delivers `drag-mouse-1`, not `mouse-1`. A button that binds only `mouse-1` ignores most taps. Both are bound.
- **A finger presses before it releases.** Emacs delivers `down-mouse-1` first, and the *standard* mode-line keymap binds it to start a window resize by dragging. If the button's map does not claim `down-mouse-1`, the stock map does, and a tap on **Next** begins resizing the pane. Bound to ignore.

- **A finger taps twice when nothing happened.** Emacs turns two presses in quick succession into double-mouse-1, whose stock mode-line binding maximises the window — the poem vanishes, replaced by a full-frame dictionary. Bound to ignore, as is the triple.
- **Terminals have no tooltips**, but the mode line has a default help-echo that Emacs would print in the echo area on hover, and iSH’s synthetic “hover” fires on every tap. The button carries a function returning nil, and the buffer sets mode-line-default-help-echo to nil.

The firstpair-reader-command text property is the second half of a defence. In some terminal decodings the click arrives with the mode-line area but *without* the string’s local map — the posn names the string, but key lookup falls through to the buffer’s mode-line bindings. So the mode map binds [mode-line mouse-1] and [header-line mouse-1] to a resolver that reads the property off the string itself, and never lets a miss fall through to Emacs’s own mode-line map:

```
(defun firstpair-reader--event-button-command (event)
  "Return the Reader button command at EVENT, or nil for a non-button cell."
  (let ((string-position (posn-string (event-start event))))
    (when (and (consp string-position)
               (stringp (car string-position))
               (integer-or-marker-p (cdr string-position)))
      (get-text-property (cdr string-position)
                          'firstpair-reader-command
                          (car string-position)))))

(defun firstpair-reader-mode-line-click (event)
  "Run the Reader button at mode-line EVENT, ignoring every non-button cell."
  (interactive "e")
  (let ((command (firstpair-reader--event-button-command event)))
    (when command
      (firstpair-reader--run-button-command event command)))))
```

And the gaps between buttons carry their own inert map, because a tap that misses a button by one cell must do *nothing* — not resize, not maximise, not print “mouse-1: select window”:

```
(defvar firstpair-reader--bar-gap-map
  (let ((map (make-sparse-keymap)))
    (dolist (area '(header-line mode-line))
      (dolist (event '(mouse-1 drag-mouse-1 down-mouse-1
                      double-mouse-1 triple-mouse-1 mouse-2 mouse-3))
        (define-key map (vector area event) #'ignore)))
    ...
  map)
  "Keymap that keeps gaps in Reader bars from invoking the stock mode line.")
```

The command runs in the window that was tapped

The last clause of the button’s docstring — “the command runs with the tapped window selected” — took a while to earn. The dictionary window has focus after a lookup. The reader taps **►** on the *book*’s mode line. Emacs runs the command in the *selected* window, which is the dictionary, and Info-next complains that the dictionary buffer is not part of a bundle. The user’s report was two words: “Langs says this buffer is not part of the FirstPair bundle”. The fix is one line in the runner:

```
(defun firstpair-reader--run-button-command (event command)
  "Select EVENT's window and invoke Reader button COMMAND."
```

```
(let ((window (posn-window (event-start event))))
  (when (window-live-p window)
    (select-window window)))
(when (commandp command)
  (call-interactively command)))
```

Once buttons selected their own window, the reader could stop duplicating bars in every pane. There had been a bar on the book and a bar on the references and a bar on the dictionary, each with the same buttons, because a button only worked in the window that had focus; the user asked, reasonably, “can we only use the middle bar?” Now there is one control bar, on the book’s mode line, and the dictionary’s own mode line carries only its lexical strip.

The three bars

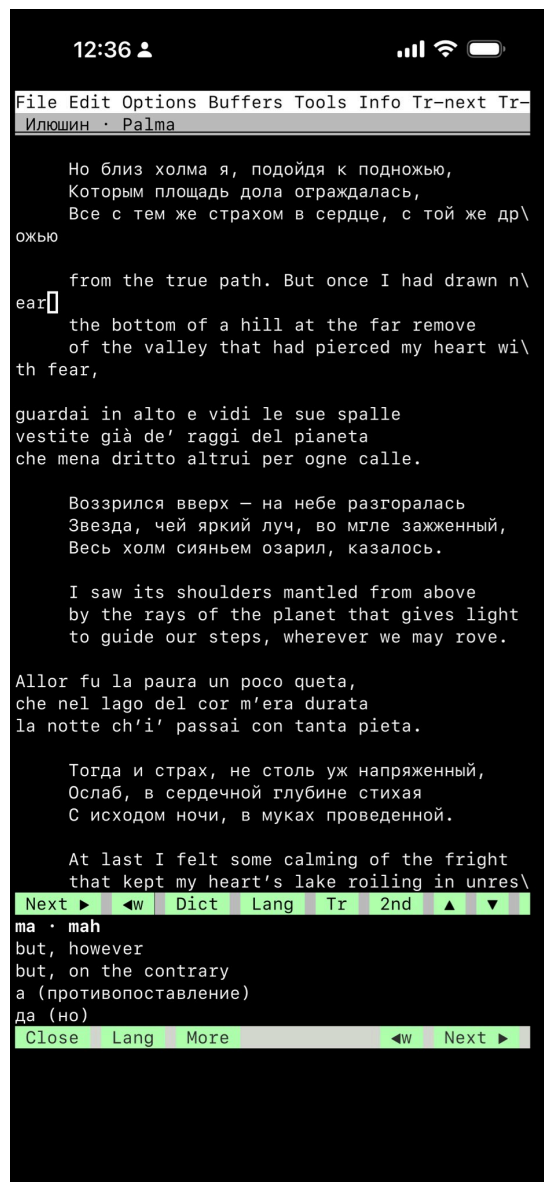


Figure 1: Dante’s Italian with English and Russian beside it, the compact dictionary, and the reader’s bars in terminal Emacs on an iPhone, during development of the touch reader.

The reader's chrome, top to bottom, on a phone:

1. **The menu bar** — Emacs's own, in the terminal, with **Tr-Eng** and **Tr-Rus** beside **File** and **Edit**.
2. **The header row** — the book window's header line: the active editions, grouped by language, each name and each ◀ a button.
3. **The poem.**
4. **The book bar** — the book window's mode line: Dict · ▲ · ▼ · ◀c · c▶ · Top · Refs · ?, followed by the node name.
5. **The references or the dictionary**, in its own window.
6. **The dictionary bar** — the dictionary window's mode line: Tr< · Tr> · 2nd · Lang at the left, and << · < · > · >> at the right, each of the four word buttons twice its natural width.

The two bars are built by `firstpair-reader--bar` and `firstpair-reader--split-bar`. The split bar is the interesting one: it puts one group of buttons at the left edge and another flush right, using the `:align-to` display specification so the right group lands at the frame's right edge whatever its width:

```
(defun firstpair-reader--split-bar (left right &optional double-right)
  "A bar with LEFT buttons at the start and RIGHT buttons at the right edge.
  When DOUBLE-RIGHT is non-nil, make each RIGHT button twice its natural width."
  (let* ((left-items ...)
        (right-items
         (butlast
          (mapcan (lambda (button)
                    (let* ((label (car button))
                          (width (and double-right
                                       (* 2 (+ (string-width label) 2))))
                      (list (firstpair-reader--button
                           label (cdr button) nil width)
                          (firstpair-reader--bar-gap))))
                  right))))
        (right-width
         (apply #'+ (mapcar (lambda (item)
                              (if (stringp item) (string-width item) 0))
                        right-items))))
    (append (cons "" left-items)
            (list (firstpair-reader--bar-gap
                  `(space :align-to (- right ,right-width))))
            right-items)))

(defun firstpair-reader--dictionary-bar ()
  "The lowest Reader bar: lexical and translation controls."
  (firstpair-reader--split-bar
   (list (cons "Tr<" #'firstpair-reader-terminal-previous-translation)
         (cons "Tr>" #'firstpair-reader-terminal-next-translation)
         (cons "2nd" #'firstpair-reader-second-translation)
         (cons "Lang" #'firstpair-lexicon-cycle-languages-command))
   (list (cons "<<" #'firstpair-reader-previous-significant-marked-lookup)
         (cons "<" #'firstpair-reader-previous-marked-lookup)
         (cons ">" #'firstpair-reader-next-marked-lookup))
```

```
(cons ">>" #'firstpair-reader-next-significant-marked-lookup))
t))
```

Why wide, and why low-right? Because the word arrows are pressed a hundred times a canto, and on a phone held in one hand the thumb rests at the bottom right. A two-column button is a miss one tap in five; a six-column button is not. And the *order* — <<, <, >, >>, significant-backward at the far left of the group and significant-forward at the far right — puts the two most used, “next word” and “next significant word”, under the thumb. The lowest bar is the nearest one; the controls a reader uses continuously are on it, at the side the hand is on, and the ones used once a canto — page, canto, Top — are on the bar above. This is the layout rule the native bar in the next chapter inherited unchanged.

Single keys

Every button is also a key, and every key is a single unmodified character, because a phone keyboard has no Control worth the name and a hardware keyboard may be attached to a device with no touchscreen:

key	action
d	dictionary for the word at point (Dict)
, .	previous / next dictionary word
j k	next / previous source word, looked up at once (> <)
J K	next / previous <i>significant</i> word, looked up (>> <<)
RET	in a source line: next word, looked up; on a link: follow it
[]	swap the edition at point for the previous / next (Tr < Tr >)
b B	add the favourite editions / keep only the favourites (2nd , hold 2nd)
t T	cycle dictionary and translation languages for- ward / backward (Lang)
v	swap the translation at point (also long press / right click)
=	report the translations showing, changing noth- ing
r g	references, glossary; n p l SPC DEL q as in Info
?	this list

RET is worth a word. Info binds it to “follow the link at point”; in a poem there are no links for pages at a time, and a reader stepping through the Italian wants the most natural key on the keyboard to mean “next word”. `firstpair-reader-return` checks, in order, that the window is the book’s, that point is

not on a link, and that the node has source regions, and only then advances; otherwise it does what Info does. Every C-c C-x chord is still there for people who have them in their fingers.

Tapping the text

A tap in the poem itself is the most common gesture of all, and it is one command:

```
(defun firstpair-reader-touch-click (event)
  "Act on a tap at EVENT: look up a dictionary word, follow a link, or move point."
  (interactive "e")
  (mouse-set-point event)
  (let ((bundle (firstpair-reader--bundle))
        (overlay (firstpair-reader--overlay-at (point))))
    (cond ((firstpair-reader--lookup-source-word-at-point bundle)
           (overlay
            (firstpair-reader-describe-word
             (buffer-substring-no-properties
              (overlay-start overlay) (overlay-end overlay))))
           ((firstpair-reader--link-at-point-p)
            (firstpair-reader-mouse-follow-nearest-node event))
           (t nil))))
```

Source word: look it up. Marked word: look it up. Link: follow it. Anything else: move point and nothing more. The poem and the dictionary are read-only buffers (`firstpair-reader-mode` sets `buffer-read-only` and turns off `text-conversion-style`, so iOS does not offer to autocorrect Dante); a tap is a query, never an edit, and never a reason to show a keyboard.

Scrolling with a finger

For a long time, swipes scrolled the terminal app's own scrollbar and not Emacs, because the app consumed them. When 1Unix learned to turn a vertical drag into wheel reports — the next chapter — the reader had to be ready for them. Terminal Emacs does not necessarily have a global binding for `mouse-4` and `mouse-5`, and it certainly does not have one that scrolls the window *under the finger* rather than the selected one. Reader 1.43 binds them locally, in the text, on both bars, and in the gaps, and scrolls one line at a time in the event's own window without changing which window is selected:

```
(defun firstpair-reader--touch-scroll (event forward)
  "Scroll EVENT's window one line FORWARD, without changing the active pane."
  (let ((window (and event (posn-window (event-start event)))))
    (when (window-live-p window)
      (with-selected-window window
        (condition-case nil
          (if forward (scroll-up-line 1) (scroll-down-line 1))
          (error nil))))))

(defun firstpair-reader--bind-touch-scroll (map)
  "Bind terminal wheel scrolling in body and chrome areas of MAP."
  (define-key map [mouse-4] #'firstpair-reader-touch-scroll-backward)
  (define-key map [mouse-5] #'firstpair-reader-touch-scroll-forward)
  (dolist (area '(header-line mode-line))
    (define-key map (vector area 'mouse-4)
      #'firstpair-reader-touch-scroll-backward))
```

```
(define-key map (vector area 'mouse-5)
  #'firstpair-reader-touch-scroll-forward)))
```

The rule the contract states is stricter than the code looks: a scroll must never look up the word beneath the finger, follow a link, resize a pane, reveal the keyboard, or move the other pane. Every one of those happened at least once during development.

What the reader cannot fix

Three things stayed broken however the Lisp was written, and each of them is why 1Unix exists.

A tap that *hides* the keyboard, then a tap on a word, brought the keyboard straight back, because iOS focus belongs to the app and the terminal view became first responder on every touch. No Emacs command can decline to be focused.

A terminal menu — **Tr-Eng** — that opened correctly but chose **None** whichever item was tapped. The reader's named commands made the items dispatchable; they did not make them *selectable*, because Emacs's TTY menu updates its active row from pointer *movement* before it accepts a release, and the terminal was not sending movement. The decisive test used no reader code at all: **File** → **Quit** opened **Find File**.

And a finger that drifted one cell between press and release, so that a tap on > released on >>, or on the gap, or on the mode-line resize handle of the pane below. Emacs dispatches a mode-line command on *release*; the release has to arrive where the press did.

Two Bars, One Contract

The controls a reader uses continuously are on the reader's lowest bar. When the terminal app grew a native button strip of its own, the question was what to put on it, and the answer — after one wrong turn — was: *the same eight buttons, sending the same eight keys, in the same order, and nothing else.*

The wrong turn

The first native strip was an input accessory view: the row iOS attaches above the software keyboard. It had arrows and a few reader commands. It disappeared whenever the keyboard did, which is to say whenever the reader was actually reading, because a full-screen poem is a poem with the keyboard hidden. And it duplicated things Emacs already had on screen, so the phone showed two rows of page and canto buttons.

The version that stayed is described in 1Unix's own contract document, and its rule is precise: the strip is *part of the terminal layout*, not an accessory, so hiding the keyboard leaves it visible; when the keyboard opens the strip moves above it; its buttons send *ordinary keyboard input* and therefore bypass terminal mouse decoding entirely; and it carries exactly the dictionary bar's controls — Tr< Tr> 2nd Lang << > >> — sending [,], b, t, K, k, j, J. Page, canto, Top, Refs, Dict, and ? are deliberately absent: they are on the book bar in Emacs, tapped once a canto, and do not need a native mirror.

This is the Objective-C, in the terminal view controller:

```
- (void)addReaderBar {
    ...
    NSArray<NSString *> *labels = @[@"Tr<", @"Tr>", @"2nd", @"Lang", @"<<", @"<",
    @">", @">>"];
    NSArray<NSString *> *keys = @[@"[", @"]", @"b", @"t", @"K", @"k", @"j", @"J"];
    NSArray<NSString *> *longKeys = @[@"[", @"[", @"B", @"T"];
    NSArray<NSString *> *accessibility = @[
        @"Previous translation", @"Next translation", @"Add favorite translations",
        @"Change language",
        @"Previous significant word", @"Previous word", @"Next word", @"Next
        significant word"
    ];
    for (NSUInteger index = 0; index < labels.count; index++) {
        BarButton *button = [[BarButton alloc] initWithFrame:CGRectZero];
        button.secondary = index < 4;
        button.accessibilityLabel = accessibility[index];
        [button setTitle:labels[index] forState:UIControlStateNormal];
        objc_setAssociatedObject(button, @selector(pressReaderKey:), keys[index],
        OBJC_ASSOCIATION_COPY_NONATOMIC);
        [button addTarget:self action:@selector(pressReaderKey:)
        forControlEvents:UIControlEventTouchUpInside];
        if (index < longKeys.count) {
            UILongPressGestureRecognizer *longPress = [[UILongPressGestureRecognizer
            alloc]
                initWithTarget:self action:@selector(pressReaderLongKey:)];
            longPress.minimumPressDuration = 0.5;
            longPress.cancelsTouchesInView = YES;
            objc_setAssociatedObject(longPress, @selector(pressReaderLongKey:),
            longKeys[index], OBJC_ASSOCIATION_COPY_NONATOMIC);
            [button addGestureRecognizer:longPress];
        }
    }
}
```

```

    }
    [bar addArrangedSubview:button];
}
...
}

- (void)pressReaderKey:(BarButton *)sender {
    NSString *key = objc_getAssociatedObject(sender, @selector(pressReaderKey:));
    if (key != nil) {
        [self pressKey:key];
    }
}
}

```

`pressKey:` is the same path a hardware keyboard takes. From Emacs’s point of view a tap on the native `>` is indistinguishable from typing `j`. That is the entire integration, and its simplicity is the point: the native strip has no idea what a translation is. It knows eight characters.

Why keys and not mouse reports

The reader’s own bar works by mouse reports, and by the time the strip was built those reports were reliable. The strip still sends keys, for three reasons.

Keys survive everything that can go wrong with mouse decoding: a lost local map, a drifted cell, a menu that ate the release. A key is a byte.

Keys survive layout. The native strip is not in the terminal grid; it has no cell to report. Its buttons are as wide as UIKit makes them — which is *wide*, wider than any mode-line button can be, and that width is why the strip is worth having at all when the same controls exist four rows up.

And keys keep Emacs in sole charge of state. If the strip sent mouse reports at some cell, the app would have to know where the reader’s buttons were. Sending `j` means the app promises nothing but “the reader binds `j`”, and the reader promises to keep binding it. That promise is the contract, and it is written down in both repositories. When the 2nd button’s meaning changed — from “toggle a second edition” in 1.19 to “add the favourites” in 1.41 — the key `b` did not change, the strip did not change, and only the accessibility label and a paragraph of documentation moved.

Long press as the inverse

Each of the four left buttons has an inverse under a half-second hold: **Tr<** held advances, **Tr>** held retreats, **2nd** held keeps only the favourites (B), **Lang** held cycles backward (T). The gesture recogniser has `cancelsTouchesInView` set so a hold sends exactly one key — the inverse — and *not* the ordinary tap on release. Emacs sees B; it never sees b first. The four word buttons have no inverses, because their inverses are their neighbours.

Toggling the native bar off

The native strip costs forty-two points of screen. A reader with a hardware keyboard, or one who prefers the Emacs bars, should be able to give the space back — and the reader in Emacs must lose nothing when they do. That is what mirroring buys: **Show Dante Reader Bar** is a persisted 1Unix setting, on by default, independent of **Keep Keyboard Hidden**, and turning it off simply removes a redundant view. The Emacs dictionary bar is still there, with the same eight controls, on the mode line.

The first implementation of the toggle could leave the terminal black. The storyboard’s constraint from the terminal’s bottom edge to the view’s bottom was a *weak* outlet; hiding the bar deactivated the terminal-to-bar constraint and reactivated the old one, and UIKit had by then released it. The repaired design keeps one strong terminal-to-bar constraint active forever and hides the strip by collapsing its height to zero; after layout, the app asks hterm to resize, invalidate, redraw, and reposition its cursor. A simulator UI test now runs two hide/show cycles and checks the terminal text after each transition. This is the kind of bug that Lisp never sees and that determines whether a feature ships.

The contract, in one table

control	Emacs bar	key	native strip	hold
previous translation at point	Tr<	[	Tr<	]
next translation at point	Tr>	]	Tr>	[
add favourite translations	2nd	b	2nd	B (favourites only)
cycle languages	Lang	t	Lang	T (backward)
previous significant word, looked up	<<	K	<<	—
previous word, looked up	<	k	<	—
next word, looked up	>	j	>	—
next significant word, looked up	>>	J	>>	—

Both columns work. Either can be turned off — the native one in Settings, the Emacs one with `firstpair-reader-touch` — and the reader is whole with either. That was the goal, and it was reached by refusing, twice, to let the native bar do anything Emacs could not.

Big Books on Small Machines

The study Dante is not large by most measures: 54 megabytes on disk, of which 26 are dictionary and 12 are Texinfo source that nobody reads. It was large enough to take *minutes* to open on an iPhone, and the reasons were instructive because none of them were the reasons we expected.

The nine-megabyte file

dante-commedia.info was 9.1 megabytes in one piece. Emacs's Info reader, on first visit to a file, inserts the *whole* file into a buffer and decodes it — a moment on a laptop, a long time on an emulated x86 core with a slow filesystem. The reader had already sharded the dictionary and indexed the regions; opening Canto I still took minutes, and it took a float-time around Info-find-node to see why.

Info has had the cure since the 1980s: *indirect* files. A large manual is a small main file holding the tag table and an Indirect: table naming subfiles, and Emacs loads only the subfile that contains the node it wants. makeinfo does this by default; our builder, which writes Info bytes itself, had not. Teaching it was a morning's work, and the code is short enough to show in full, because the format is worth knowing:

```
if len(body) - preamble_length > self.split_bytes and len(node_starts) > 1:
    # Split at node boundaries into subfiles of about `split_bytes`.
    # Tag offsets stay those of the single file; each subfile's
    # Indirect offset is the single-file position of its first node,
    # which is how Emacs's Info-read-subfile locates a node.
    groups: list[list[tuple[str, int]]] = [[]]
    for index, (name, start) in enumerate(node_starts):
        end = node_starts[index + 1][1] if index + 1 < len(node_starts) else
len(body)
        current = groups[-1]
        current_size = (end if not current else end - current[0][1])
        if current and current_size > self.split_bytes:
            groups.append([(name, start)])
        else:
            current.append((name, start))
    indirect_lines = []
    for number, group in enumerate(groups, 1):
        start = group[0][1]
        end = (groups[number][0][1] if number < len(groups) else len(body))
        subfile_name = f"{self.manual.filename}-{number}"
        header = f"This is {subfile_name}, produced by {self.produced_by}.
\n\n".encode("utf-8")
        subfiles.append((subfile_name, header + body[start:end]))
        indirect_lines.append(f"{subfile_name}: {start}")
    main = (
        pieces[0]
        + SEPARATOR + "\nIndirect:\n" + "\n".join(indirect_lines) + "\n"
        + SEPARATOR + "\nTag Table:\n(Indirect)\n" + "\n".join(tag_rows) + "\n"
        + SEPARATOR + "\nEnd Tag Table\n" + trailer
    )
```

The subtlety is in the comment. Tag-table offsets in an indirect manual are positions in the *virtual* single file — as if the subfiles were concatenated after the preamble — and each Indirect: line records the virtual offset at which that subfile's nodes begin. When Emacs wants a node at virtual position *p*,

Info-read-subfile walks the Indirect: table for the last entry whose offset is not greater than p , reads that subfile, skips its own header to the first `^_`, and adds p minus the subfile's offset. Here is the walk, from Emacs's `info.el`:

```
(defun Info-read-subfile (nodepos)
  ;; NODEPOS is either a position in bytes (in the Info file as a whole,
  ;; not relative to a subfile) or the name of a subfile.
  (let (lastfilepos
        lastfilename)
    (if (numberp nodepos)
      (with-current-buffer (marker-buffer Info-tag-table-marker)
        (goto-char (point-min))
        (or (looking-at "\\^_")
            (search-forward "\\n\\^_"))
        (forward-line 2)
        (catch 'foo
          (while (not (looking-at "\\^_"))
            (if (not (eolp))
              (let ((beg (point))
                    thisfilepos thisfilename)
                (search-forward ": ")
                (setq thisfilename (buffer-substring beg (- (point) 2)))
                (setq thisfilepos (read (current-buffer)))
                (forward-line 1)
                (if (> thisfilepos nodepos)
                  (throw 'foo t))
                (setq lastfilename thisfilename)
                (setq lastfilepos thisfilepos))
              (forward-line 1))))
          (setq lastfilename nodepos)
          (setq lastfilepos 0))
      (or (equal Info-current-subfile lastfilename)
        (let ((inhibit-read-only t))
          (setq buffer-file-name nil)
          (widen)
          (erase-buffer)
          (info-insert-file-contents lastfilename)
          (set-buffer-modified-p nil)
          (setq Info-current-subfile lastfilename)))
        ...))
```

Two things to notice for later. The subfile is read with `info-insert-file-contents` — lower-case *i* — and the buffer is *erased* when the subfile changes. Both facts matter to the chapter on translations, where the reader edits the buffer.

The result: a 5.5-kilobyte main file, thirty-seven subfiles of about 250 kilobytes each, and Canto I opening in 0.13 seconds on the laptop where it had taken 0.7, with the whole-file read — the part that was minutes on the phone — gone entirely. The validator reassembles the virtual stream and checks every Indirect: offset; a new test opens a node straight from a subfile in batch Emacs and asserts which subfile Emacs chose.

Regions by byte offset

Registering a bundle used to parse `regions.tsv` whole: 56,362 rows for the study Dante, 5 megabytes, a second on a laptop, three minutes on the phone — *before the first page*. The rows are only ever wanted one node at a time. So the builder groups them by node and writes `regions.index.json`, a map from `manual<TAB>node` to a byte range, and the reader reads a range:

```
(defun firstpair-bundle-regions-for-node (bundle manual node)
  "Return the aligned-text regions recorded for NODE of MANUAL in BUNDLE.
  Read from the bundle's byte index on first use, or, for a bundle without an
  index, from the whole table parsed once."
  (let* ((table (firstpair-bundle-regions bundle))
        (key (concat manual "\0" node))
        (cached (gethash key table 'none)))
    (if (not (eq cached 'none))
        cached
        (let* ((file (expand-file-name "data/regions.tsv" (firstpair-bundle-root
bundle)))
              (span (gethash (concat "index\0" key) table))
              (rows (cond (span
                          (with-temp-buffer
                            (insert-file-contents file nil (elt span 0) (elt span 1))
                            (firstpair-bundle--parse-region-rows (buffer-string))))
                        ((and (not (gethash "whole\0loaded" table)) (file-readable-p
file))
                          ;; Older bundle: parse everything once into the cache.
                          (puthash "whole\0loaded" t table)
                          (maphash (lambda (k v) (puthash k v table))
                                   (firstpair-bundle--regions file))
                          (gethash key table))
                        (t nil))))))
          (puthash key (nreverse rows) table)
          (gethash key table))))))
```

`insert-file-contents` with `BEG` and `END` is the whole trick — Emacs reads exactly the bytes asked for — and the `older-bundle` branch means a reader can open a bundle built before the index existed. Registration fell from 210 milliseconds to one, and the phone's “forever” to a blink.

Sharded tables

The dictionary had the same disease. The first lookup parsed a 30-megabyte gloss table and a 2-megabyte forms table into hash tables, and on the phone that was the hang the user reported as “c-c c-d is hanging in ish now” — followed, a while later, by “yeah came to”. Both tables now ship as one file per first letter, and a lookup loads one letter:

```
(defun firstpair-lexicon--gloss-shard (key)
  "The gloss shard KEY lives in: its first letter, lowercased, or \"_\"."
  (let ((first (and (> (length key) 0) (downcase (substring key 0 1)))))
    (if (and first (string-match-p "[[:alpha:]]" first)) first "_"))

(defun firstpair-lexicon--forms-table (bundle form)
  "The forms table holding FORM.
  Its first-letter shard, or the single table of an older bundle."
```

```
(let* ((root (firstpair-bundle-root bundle))
      (shard (concat "forms/" (firstpair-lexicon--gloss-shard form) ".tsv")))
  (cond ((file-readable-p (expand-file-name (concat "lexicon/" shard) root))
        (firstpair-lexicon-table bundle shard))
        ((file-readable-p (expand-file-name "lexicon/forms.tsv" root))
         (firstpair-lexicon-table bundle "forms.tsv"))
        (t (make-hash-table :test #'equal)))))
```

The study Dante ships 24 forms shards (2.2 megabytes together) and 25 gloss shards (23 megabytes); the letter *s* is the largest, and it loads in well under a second on the phone. Because even that is perceptible, the first lookup shows Loading dictionary., . . . in the echo area while it parses and then Dictionary ready: WORD; later lookups are silent. And a table that has been parsed is a hash table in the bundle’s tables cache for the life of the session.

The numbers

For the study Dante, before and after, on a 2024 laptop:

operation	before	after
register bundle	210 ms	1 ms
open Canto I (cold)	0.40 s read + 0.33 s goto	0.13 s
first lookup	parse 32 MB	parse one letter
dante-commedia.info	9.1 MB, one file	5.5 KB + 37 subfiles

On the phone the ratios are far larger, because the cost that dominated — reading and decoding large files through an emulated kernel — is the one that was removed rather than reduced. The lesson generalises beyond phones: *a table a laptop parses in a second must be assumed to take minutes somewhere else*, and the fix is always the same — make the first page depend on nothing large.

Reordering in place

The chapter on translations promised to explain how the header row’s ◀ actually moves a translation earlier when the order is baked into the file. It moves the text.

Each unit of the poem is written as the Italian block, a blank line, then every translation block in build order, each followed by a blank line. Hiding is overlays; but the order of visible blocks is the order in the buffer, and the reader wants the order the user chose. So on every refresh, for each unit, the reader computes the order the state variables imply, and if the buffer disagrees, rewrites that unit’s translation blocks — and *only* them:

```
(defun firstpair-reader--order-regions (bundle)
  "Rearrange each unit's translation blocks into the chosen display order.
  Blocks move bodily within their unit — the source lines never move, so
  the lexicon's marked positions stay exact. The rows returned by
  `firstpair-reader--regions' are updated in place. A unit whose blocks
  are not separated by single blank lines is left untouched."
  (when (and (firstpair-bundle-translations bundle)
            (equal (firstpair-bundle-manual) (firstpair-bundle-reader bundle)))
    (let ((units (make-hash-table :test #'equal)))
      (order nil)))
```

```

(dolist (region (firstpair-reader--regions bundle))
  (unless (plist-get region :source)
    (let ((unit (plist-get region :unit)))
      (unless (gethash unit units) (push unit order))
      (push region (gethash unit units))))))
(let ((inhibit-read-only t))
  (dolist (unit (nreverse order))
    (let* ((blocks (sort (gethash unit units)
                        (lambda (a b)
                          (< (plist-get a :start) (plist-get b :start))))
           (desired (firstpair-reader--desired-block-order bundle blocks)))
      (when (and (cdr blocks)
                  (not (equal blocks desired))
                  (firstpair-reader--contiguous-blocks-p blocks))
        (let ((texts (mapcar (lambda (region)
                                (cons region (firstpair-reader--region-lines
region)))
                                desired))
              (line (plist-get (car blocks) :start)))
          (save-excursion
            (goto-char (point-min))
            (forward-line (1- line))
            (let ((start (point)))
              (goto-char (point-min))
              (forward-line (plist-get (car (last blocks)) :end))
              (delete-region start (point))
              (goto-char start)
              (insert (mapconcat (lambda (pair)
                                  (mapconcat #'identity (cdr pair) "\n"))
                                texts "\n\n"))
              "\n"))))
          (dolist (pair texts)
            (let ((region (car pair))
                  (size (length (cdr pair))))
              (plist-put region :start line)
              (plist-put region :end (+ line size -1))
              (setq line (+ line size 1))))))))))

```

Three invariants make this safe rather than reckless.

The source never moves. Translation blocks are permuted within their unit, after the Italian and before the next unit's Italian. A unit's total line count is unchanged by a permutation, so every later line in the node keeps its number, and `marked.tsv` positions and every other unit's region rows stay exact.

Blocks must be contiguous. `--contiguous-blocks-p` checks that each block begins two lines after the previous one ends — one blank line between — and a unit that fails the check is left alone. The builder always writes them that way; the check is there for the day it does not.

Region rows follow the text. `firstpair-reader--regions` hands out *copies* of the bundle's rows, cached per node in the buffer, and `--order-regions` updates their `:start` and `:end` in place. Everything

else — hiding, source spans, the target under point — reads through that cache and sees true line numbers.

And one invalidation, which is where `info-insert-file-contents` comes back. When Emacs leaves this subfile for another and returns, it erases the buffer and re-reads pristine bytes; the cached rows would then describe an order the buffer no longer has. So the cache is stamped with a per-buffer generation counter, bumped by advice on the function Info actually calls:

```
(defvar-local firstpair-reader--subfile-generation 0
  "Bumped whenever Info re-reads this buffer's file, invalidating regions.")

(defun firstpair-reader--note-file-read (&rest _arguments)
  "Record that Info replaced the buffer text with freshly read pristine bytes."
  (setq firstpair-reader--subfile-generation (1+ firstpair-reader--subfile-
generation))
  (setq firstpair-reader--node-regions nil))

;; in firstpair-reader-register:
(advice-add 'info-insert-file-contents :after #'firstpair-reader--note-file-read)
```

A confession belongs here, since the book promised them. Reader 1.39 shipped that `advice-add` naming `Info-insert-file-contents`, with a capital *I*. Emacs symbols are case-sensitive; no such function exists; `advice-add` on an undefined function succeeds silently and waits for a definition that never comes. The cache was therefore never invalidated, and a reader who reordered translations, paged into the next subfile, and came back would have seen the wrong lines hidden. It was found while writing this chapter, by reading `Info-read-subfile` closely enough to quote it, and fixed in 1.47 with a test that asserts the advice is attached to the function Info calls. Quote your dependencies; it is the cheapest code review there is.

Resume, Package, Verify

A reader on a phone is not opened; it is resumed. And a reader that updates itself must be able to prove what it installed. This chapter is the plumbing that makes the reader a thing one can rely on rather than a demonstration.

Where you were

The reader saves its state on every node change, whenever Emacs has been idle five seconds, and at exit. The state is a plist per bundle, keyed by the bundle's real directory, written with pp so a person can read and edit it:

```
(defun firstpair-reader-save-state ()
  "Remember where reading stopped in the current bundle, and what was shown."
  (interactive)
  (let ((bundle (firstpair-bundle-current)))
    (when (and firstpair-reader-resume bundle firstpair-reader-mode (not firstpair-
reader--setting-up)
              (equal (firstpair-bundle-manual) (firstpair-bundle-reader bundle))
              Info-current-node)
      (let* ((key (firstpair-reader--state-key bundle))
              (state (list :node Info-current-node :point (point)
                           :languages firstpair-lexicon-languages
                           :selections firstpair-reader-translation-selections
                           :language-order firstpair-reader-language-order
                           :saved (format-time-string "%FT%T%z"))))
        (states (firstpair-reader--load-states)))
        (setf (alist-get key (cdr states) nil nil #'equal) state)
        (with-temp-file firstpair-reader-state-file
          (insert ";; FirstPair Reader: where each bundle was left. Generated; edit
freely.\n")
          (pp states (current-buffer)))))))
```

firstpair-reader--setting-up is bound to t while firstpair-read opens the Top nodes of both manuals, so the act of opening does not overwrite the place being returned to. The idle timer is armed once; the kill-emacs-hook is wrapped in ignore-errors because nothing about quitting should ever fail on account of a bookmark.

Resuming reads the plist back, and here the slot-to-list migration lives — old files hold :choices and :seconds; new ones hold :selections; both are honoured:

```
(defun firstpair-reader--saved-selections (state)
  "Return ordered translation selections from current or legacy STATE.
Reader 1.39 and later store `:selections'. Readers 1.38 and 1.40 stored one
`:choices' entry and an optional `:seconds' entry per language; preserve both
as checked editions when upgrading."
  (or (copy-tree (plist-get state :selections))
      (let (selections)
        (dolist (pair (plist-get state :choices) (nreverse selections))
          (push (cons (car pair)
                      (delete-dups
                       (delq nil
                            (list (cdr pair)
                                   (alist-get (car pair) (plist-get state :seconds)
                                              nil nil #'equal))))
                selections))))))
```

```

nil nil #'equal))))))
selections))))))

```

Which reader runs

A bundle ships its own copy of the three Lisp files, so a reader who has never installed anything can (load ".../init.el") and go. A reader who *has* installed the firstpair-reader package wants the package — it may be newer. And a package that is *older* must never shadow a bundle’s corrected reader. The bundle’s init.el, generated by the builder with the reader version baked in, decides:

```

(let ((bundle (file-name-directory (or load-file-name buffer-file-name)))
      (minimum-version (version-to-list "1.47")))
  ;; Interactively an installed firstpair-reader package wins when it is at
  ;; least as new as this bundle: activate packages if that has not happened
  ;; yet, and otherwise use the bundled reader. In batch (validation, scripts)
  ;; the bundle's own Lisp is used, whatever is installed.
  (unless noninteractive
    (when (and (fboundp 'package-initialize) (not (bound-and-true-p package--
initialized)))
      (ignore-errors (package-initialize))))
  (if (and (not noninteractive)
          (fboundp 'package-installed-p)
          (package-installed-p 'firstpair-reader minimum-version)
          (locate-library "firstpair-reader"))
      nil
      (add-to-list 'load-path (expand-file-name "lisp" bundle)))
  (require 'firstpair-reader)
  (firstpair-reader-register bundle))

```

The batch clause was a bug fix before it was a design. Validation runs emacs --batch against the bundle’s own Lisp; a machine with an old package installed validated the wrong reader, and the failure — firstpair-lexicon-forms-tables void — pointed at a function the bundle’s Lisp had and the package did not. The interactive clause was a bug fix too: a phone with reader 1.9 installed as a package ran 1.9 while its bundle’s init.el reported 1.14, because load-path order let the package win.

The package and its sidecar

firstpair-emacs package assembles firstpair-reader-<version>.tar from the same three files plus a -pkg.el and the handbook compiled as the package’s own Info manual — the reader documents itself in the format it reads. The version is read from the ;; Version: header, and the packager refuses to build if the three files disagree.

The public copy lives at a stable URL, firstpair.org/emacs/firstpair-reader.tar, deliberately *unversioned* so a phone can fetch it with one short command; the user’s instruction was “keep site file name unversioned for now so it’s faster to download”. Beside it is a sidecar, firstpair-reader.tar.sha256:

```

# version 1.47
79a073ec82f5e734f38e548d2d649e41710f636e6900a7bd5e753f8b9b02dacb  firstpair-
reader.tar

```

The bundle’s launcher, dante.sh, fetches the sidecar first — 120 bytes. If the version it names is already installed, nothing else is downloaded and Emacs opens the book. If not, it downloads the tar, verifies

the digest, verifies that the archive really is `firstpair-reader` at the declared version, installs it, and only then starts Emacs. It refuses to run at all while another Emacs is alive, because replacing package files under a live session is how unsaved buffers are lost. If the network is absent, a failed fetch falls back to the installed reader and the local book — offline reading is a convenience; but a wrong hash is fatal, because accepting unverified code is not.

Proving it works: `firstpair-check`

The validator's last stage runs Emacs itself. `firstpair-check.el` — loaded by `firstpair-emacs` `validate`, never shipped in a bundle — opens every node of both manuals with the reader active and counts:

```
(defun firstpair-check (root nodes-file)
  "Verify the bundle at ROOT by visiting every node listed in NODES-FILE."
  (let* ((bundle (firstpair-check--bundle root))
        (nodes (firstpair-bundle--read-json nodes-file))
        (visited 0) (expected-marks 0) (located-marks 0)
        (missing nil) (unresolved nil) (references 0) (lexicon-failures nil)
        (lexicon-tested 0))
    (dolist (manual nodes)
      (let ((stem (symbol-name (car manual))))
        (dolist (node (cdr manual))
          (with-current-buffer (firstpair-check--visit stem (format "(%s)%s" stem
node))
            (unless firstpair-reader-mode
              (error "Reader mode is not active in %s node %s" stem node))
            (cl-incf visited)
            (let ((expected (length (firstpair-bundle-marked-words bundle stem
node)))
                  (found (length (firstpair-reader--marked-overlays))))
              (cl-incf expected-marks expected)
              (cl-incf located-marks found)
              (unless (= expected found)
                (push (format "%s:%s expected %d located %d" stem node expected
found) missing))))))
          ...
        (princ (json-encode
          `((visited . ,visited)
            (expectedMarks . ,expected-marks)
            (locatedMarks . ,located-marks)
            (missingMarks . ,(vconcat (nreverse missing)))
            (references . ,references)
            (unresolved . ,(vconcat (nreverse unresolved)))
            (lexiconTested . ,lexicon-tested)
            (lexiconFailures . ,(vconcat (nreverse lexicon-failures))))))
        (terpri)))
```

Every marked word must be located; every recorded cross-reference must resolve; every language that has any glosses must render at least one; twenty-five forms must analyse and render. For the study Dante that is 132 nodes and zero failures, and it runs on every build. The Python side of the validator parses both Info files structurally (following `Indirect: tables`), compiles the Texinfo with `makeinfo`

when present, checks the sealed inventory's digests, and refuses a bundle whose sourceCommit is not HEAD.

Testing Emacs from Python

The framework's tests are Python unittest cases that build a small fixture bundle from JSON and then drive emacs --batch -Q --eval with a Lisp script, capturing what it prints. This is unglamorous and it works: thirty-three tests, fifteen seconds, and every one of the behaviours in this book — redirecting references, locating marks, checkbox menus, feedback hooks, in-buffer reordering, resume, the touch bars' keymaps — has a script that exercises it and a string it must print. One idiom is worth stealing: to test a TTY menu, the script walks the keymap with map-keymap, finds the item by label, and call-interactively its command, then fires post-command-hook by hand and captures the message. To test a mode-line tap, it constructs the event: (firstpair-reader-mode-line-click '(mouse-1 nil)) with mouse-set-point stubbed.

One gotcha for anyone doing the same: format-mode-line returns the empty string in batch Emacs, always. The tests that check the header row flatten the format list themselves. An afternoon was spent discovering that.

1Unix

Everything in the *Touch* chapter ends at the same wall: the terminal. Emacs can decode a mouse report; it cannot make a phone send one at the right place, and it cannot stop iOS from raising a keyboard. So the terminal was forked.

iSH is an astonishing program: a user-mode x86 emulator and a Linux system-call layer, running Alpine Linux inside an ordinary iPhone app, with hterm — the terminal from Chrome OS’s Secure Shell — as its screen. Emacs 27 runs in it. The fork, built for one purpose and named 1Unix, keeps the emulator untouched and changes the two thin layers that a finger passes through: the JavaScript around hterm, and the Objective-C view controller that owns the keyboard and the layout. This chapter is what changed, in the order the problems appeared.

A finger becomes a mouse

hterm turns DOM mouse events into terminal mouse reports when a program has requested them. iOS synthesises *compatibility* mouse events from touches — but on iPhone those synthetic events can lose the finger’s position and land at the top row of the terminal, or on the focused element rather than the screen. The symptom on the reader was the one described earlier: every menu chose its first item, and a tap on a word in the fifth tercet looked up a word in the first.

1Unix stops trusting the compatibility events entirely. It listens for touchstart, touchmove, and touchend on hterm’s screen node in the capture phase, converts each touch’s own viewport coordinates into the mouse events hterm expects, and suppresses the synthetic ones for a second after every touch — on the *document*, not the screen, because hterm also listens on the document and on its cursor node, and iOS may target any of them:

```
const suppressCompatibilityMouse = (e) => {
  if (performance.now() >= suppressCompatibilityMouseUntil) return;
  if (e.type !== 'mousemove') {
    native.log({
      event: 'suppressed-compatibility-mouse',
      type: e.type, clientX: e.clientX, clientY: e.clientY,
    });
  }
  e.preventDefault();
  e.stopImmediatePropagation();
};

['mousedown', 'mouseup', 'mousemove', 'click'].forEach((type) => {
  // hterm listens on the whole iframe document as well as the screen and
  // cursor. iOS may target its compatibility event at any of those nodes.
  term.getDocument().addEventListener(type, suppressCompatibilityMouse,
    {capture: true});
});
```

The first version suppressed on the screen element only. It worked in the simulator and failed on the phone, where the events arrived at the document. “Fix document touch suppression” is a one-line commit that took a day.

DECSET 1003, which hterm did not have

Emacs’s xterm-mouse-mode requests mode 1003, *all-motion* tracking, so that a TTY popup menu can move its highlighted row as the pointer moves and act on the row under the pointer at release. hterm

1.91 does not implement 1003. It silently ignored the request, Emacs got press and release with no movement between, and the menu — whose active row is updated only by movement — acted on its initial row. **File** → **Quit** opened **Find File**. **Tr-Rus** → **Лозинский** chose **None**.

The fix treats 1003 as hterm’s drag-tracking mode and lets the touch handlers supply the movement:

```
// This hterm predates DECSET 1003 (all-motion mouse tracking). TTY menus in
// Emacs use that mode to update the active row before accepting a click. Treat
// it as drag tracking, and let the touch handlers distinguish a stable-cell tap
// from a vertical wheel gesture.
let terminalTouchAllMotion = false;
const originalSetDECMODE = term.vt.setDECMODE.bind(term.vt);
term.vt.setDECMODE = function(code, state) {
  if (parseInt(code, 10) == 1003) {
    terminalTouchAllMotion = state;
    this.mouseReport = state ?
      this.MOUSE_REPORT_DRAG : this.MOUSE_REPORT_DISABLED;
    this.terminal.syncMouseStyle();
    return;
  }
  originalSetDECMODE(code, state);
};
```

With that, a touch-down sends a *movement* report at its cell before the press, and the menu’s active row is the row the finger is on.

The atomic tap

The next problem is the one the reader’s drag-mouse-1 bindings were fighting from the other side. Emacs dispatches a mode-line command on *release*, and a finger drifts between press and release — on a narrow button that is the difference between > and >>, or between a button and the resize handle under it. Emacs cannot know where the finger *started*. The terminal can.

So in mode 1003, 1Unix does not report the press when the finger goes down. It waits. If the finger lifts without having moved more than three-quarters of a line, it emits movement, press, and release *together, all at the touch-down cell*, and discards the physical release position:

```
const reportPendingTouchTap = () => {
  if (!terminalTouchPendingTap || terminalTouchStartPoint == null) return;
  // Keep all three reports on the touch-down cell. Physical finger drift must
  // not move a narrow Reader or TTY-menu target before release.
  term.vt.lastMouseDragResponse_ = null;
  reportTouchAsMouse('mousemove', terminalTouchStartPoint, 1);
  reportTouchAsMouse('mousedown', terminalTouchStartPoint, 1);
  reportTouchAsMouse('mouseup', terminalTouchStartPoint, 0);
  terminalTouchPendingTap = false;
};
```

This is build 818, and it is the change that made the reader’s bars trustworthy: a single tap on **Next** advances once, and never resizes a pane or opens a *Messages* buffer. Other mouse modes keep ordinary press-drag-release semantics, because programs that use them — a mouse-drag-region in a plain buffer — need the drag.

A drag becomes a wheel

Deferring the press had a second benefit: the terminal now knows, at touchmove, whether a touch is a tap or a scroll. Movement beyond the threshold turns the touch into wheel steps, delivered at the *touch-down* cell so Emacs scrolls the pane the gesture began in, in the natural direction:

```
const reportTouchScroll = (touch) => {
  if (!terminalTouchPendingTap || terminalTouchStartPoint == null ||
      terminalTouchLastY == null) return;
  terminalTouchScrollRemainder += terminalTouchLastY - touch.clientY;
  terminalTouchLastY = touch.clientY;
  const lineHeight = term.scrollPort_.characterSize.height || 16;
  const step = Math.max(8, lineHeight * 0.75);
  const direction = Math.sign(terminalTouchScrollRemainder);
  const count = Math.min(20,
    Math.floor(Math.abs(terminalTouchScrollRemainder) / step));
  if (count == 0) return;
  terminalTouchScrolling = true;
  for (let i = 0; i < count; ++i) {
    // Finger up moves the text up (wheel down), as in a native scroll view.
    reportTouchAsWheel(terminalTouchStartPoint, direction);
  }
  terminalTouchScrollRemainder -= direction * count * step;
};
```

hterm turns a synthetic WheelEvent into mouse-4/mouse-5 reports, and the reader's local bindings from the *Touch* chapter scroll that window one line per step. A scroll that ends is not a tap: touchend checks terminalTouchScrolling and reports nothing. The whole state machine — tap, drift, scroll, two-finger — is tested in Node against a copy of the implementation sliced out of term.js, with fake TouchEvents, before every signed build.

The keyboard that would not stay down

With taps arriving correctly, the last obstacle was iOS itself. Hiding the software keyboard gave the poem the whole screen; the next tap made the terminal view first responder and the keyboard slid straight back. This was not fixable from the JavaScript side. 1Unix adds a persisted setting, **Keep Keyboard Hidden**, in the app's own preferences: when a program has enabled mouse reporting and the switch is on, a single-finger tap remains an application mouse event and the native focus request is suppressed. A *two-finger* tap explicitly restores the keyboard:

```
term.scrollPort_.screen_.addEventListener('touchstart', (e) => {
  if (terminalMouseReporting() && e.touches.length == 2) {
    suppressCompatibilityMouseUntil = performance.now() + 1000;
    ...
    native.focus({force: true});
  } else if (terminalMouseReporting() && e.touches.length == 1 &&
    e.changedTouches.length == 1) {
    ...
    terminalTouchPendingTap = terminalTouchAllMotion;
    e.preventDefault();
    e.stopImmediatePropagation();
    if (!terminalTouchPendingTap) {
      reportTouchAsMouse('mousedown', terminalTouchPoint, 1);
    }
  }
});
```

```

        terminalTouchPressed = true;
    }
    native.focus({mouseReporting: true});
}
}, {capture: true, passive: false});

```

Outside mouse mode nothing changes: a tap focuses the terminal and shows the keyboard, as any terminal should. This generic change — no reader, no Dante, just “a terminal in mouse mode should not raise the keyboard on every tap” — was proposed upstream to iSH as a pull request; the personal build carries it plus everything book-specific on its own branch.

The native strip, revisited from this side

The previous chapter described the Reader strip from Emacs’s point of view: eight labels, eight keys, four inverses. From the app’s side the only subtle part is *where* it lives. An `inputAccessoryView` disappears with the keyboard. The strip is instead a view in the terminal’s own layout, constrained to the terminal’s bottom edge; when the keyboard appears the strip’s bottom constraint moves above it, and when the keyboard hides the strip stays. A `BarButton` with secondary styling for the four translation controls and primary styling for the four word controls makes the thumb’s targets the brighter ones.

And the toggle. **Show Dante Reader Bar** is its own persisted preference. Off gives the terminal the strip’s height at once; on restores it without restarting Emacs. The lesson recorded in the contract document is a UIKit one: the storyboard’s terminal-bottom constraint is a weak outlet that UIKit may release once deactivated, so the implementation never switches constraints — it keeps the terminal-to-strip constraint active permanently and collapses the strip’s height to zero — and after layout it asks `hterm` to resize, invalidate, redraw, and reposition the cursor, or the terminal comes back black. Two hide/show cycles run in a simulator test before every signed build.

Fonts, cells, and first paint

One more class of bug had nothing to do with touch and everything to do with where a tap lands. `hterm` computes its cell grid from the measured size of a character; if the bundled terminal font has not finished loading when `hterm` measures, the grid is computed for the fallback font, the visible glyphs are drawn in the real one, and every mouse report is off by a column or two. 1Unix waits for the selected face to load, remeasures and redraws, and disables kerning and ligatures — `PragmataPro`’s ligatures were changing the width of `->` — so that glyphs, cursor, and mouse cells share one fixed grid. The terminal stays hidden during bootstrap and is revealed only after font, colours, and cursor have been applied atomically; a launch that exposed the transparent bootstrap state is a test failure.

The phone check

Every signed build is installed on a real phone and walked through a short protocol, written down in the contract document and worth repeating because it is the whole architecture in eight steps:

1. Open Dante in Emacs with terminal mouse mode enabled.
2. Hide the keyboard and tap **Next word**, **Previous word**, and the translation controls. The keyboard must remain hidden.
3. Tap a word in the poem. The dictionary must update without showing the keyboard.
4. Open the Emacs **File** menu and choose an item below the first row. The tapped item must run.
5. Open **Tr-Eng** or **Tr-Rus** and choose a translation below **None**. The tapped translation must appear.
6. Type across a full line in `PragmataPro`. The cursor must stay on the next cell, without drift.

7. Force-quit and reopen repeatedly. Every launch must show the same crisp, full-contrast font.
8. Two-finger tap the terminal. The keyboard must appear.

Step four is the one to run first when anything is wrong, because it contains no reader code at all. If **File** → **Quit** opens **Find File**, the terminal has lost the touched cell or the movement report, and no amount of Lisp will help.

What the fork is and is not

1Unix is not a new terminal. It is iSH with perhaps four hundred lines changed in the JavaScript around hterm, a few hundred in the view controller and preferences, a name, an icon, a bundle identifier so it can live beside stock iSH, and a document. Its bundle identifier is `net.hurz.danteish`; reinstalling a build over it preserves the Alpine container — packages, books, dotfiles — and a Personal Team signing profile lasts seven days, so the signed build is repeated weekly. Nothing in it knows what a translation is. It knows that a tap should land where the finger did, that a menu needs a movement report, that a drag is a scroll, that the keyboard should stay away until asked for, and that eight buttons send eight keys. Those are properties of a terminal, not of a book, and the useful ones have been offered back.

What We Learned

The headboard of this book draws the pipeline as a chain of illuminated pages — compose, copy, transport, compile, link, navigate — from a scriptorium in the Dolomites to a phone in Venice. The chain is a fair picture of the architecture, and each link taught something. Here they are, for anyone who wants to build a reader of their own.

Info is still a good hypertext. Named nodes, cross-references between manuals, a directory, a reader that runs everywhere, a format you can write by hand. A book that is a graph fits it better than it fits HTML, because Info has no layout to fight and every address is a string.

Write the bytes yourself when positions matter. `makeinfo` is right for manuals. A reader that underlines by line and column, hides by line range, and permutes blocks needs a writer that knows what it wrote. The Texinfo source still ships, and `makeinfo` still compiles it in validation, so the two renderings police each other.

Roles are window parameters. `set-window-parameter` with a symbol, and a finder that walks `window-list`, is all the window management a three-pane reader needs — plus a way to borrow a pane and give it back.

One let of dynamic scope beats a global. The cross-reference redirect is an `:around` advice that consults a variable bound only while a reader command runs. Everything Info does on its own is untouched.

Overlays for hiding; text edits only when order matters — and then with invariants. Hiding a translation is an invisible overlay with `evaporate`. Reordering has to move text, so it moves only translation blocks, only within their unit, only when they are contiguous, and it keeps its own copy of the row table in step. And it invalidates that copy on the function Info *actually* calls to re-read a file — checked by name, in a test.

Nothing large before the first page. Byte-indexed region tables read with `insert-file-contents` `BEG` `END`; first-letter shards; indirect Info files. The measure is not the laptop's; it is the slowest machine a reader will hold.

A control that cycles hides state; a control that toggles shows it. Checkboxes and a header row that is the state replaced two slots and three rotate commands, and the reader got simpler as it got more capable. One additive button with a `hold-for-inverse` — 2nd — covers the case a toggle model handles badly: “bring back my favourites, touch nothing else”.

A button is a propertized string with a keymap, and a finger is not a mouse. Bind `drag-mouse-1` as well as `mouse-1`; claim `down-mouse-1` and the double and triple clicks; give the gaps an inert map; suppress the help echo; resolve the command from a text property when the terminal loses the string's map; run the command in the window that was tapped. Each of those is a phone bug with a name.

Named commands for menus. A terminal host's menu bridge may not carry an anonymous closure. intern a name and `fset` it.

Post feedback from post-command-hook. A TTY menu erases the echo area after your command has run. Say it later, once, and remove the hook.

Keys are the most robust IPC there is. The native strip sends `j`. The reader binds `j`. That is the entire contract between an Objective-C view and an Emacs minor mode, and it survived a change in what the button means.

Trust nothing between the finger and the cell. Suppress synthetic mouse events at the document. Emulate the mouse mode Emacs asks for. Report movement before the press. Make a tap atomic at the touch-down cell. Test **File** → **Quit** before touching your own code.

Version the reader, verify the package, and let the bundle refuse an older one. `;; Version:` headers that must agree, a sidecar with a digest, a launcher that checks it, and an `init.el` that prefers the installed package only when it is at least as new.

Quote your dependencies. The one real bug found while writing this book was found by pasting Info-read-subfile into a chapter and reading it. Reading Emacs's source, closely, is the cheapest review there is.

And test in batch. `emacs --batch -Q --eval`, a script that `princs`, a Python test that reads it. Thirty-three of those run in fifteen seconds and cover every behaviour in this book. Just remember that `format-mode-line` returns nothing there.

Making your own

The reader is not tied to Dante. A bundle is declared by an `emacs` block in a title's `vault.build.json` — the reader pages, the record sets, the lexicon language and its translation sources — and built by `firstpair-emacs build`. A prose title with a Latin lexicon (Cicero), a code title whose chapters open the delivered source files (the Sail Rust Book), and an aligned poem with nine translations (Dante) are all the same three Lisp files with different data. The delivery contract, `EMACS-DELIVERY.md`, is the specification; the handbook, which the package installs as its own Info manual, is the user's guide; and everything quoted here is in `publishing/emacs/lisp/` of the FirstPair repository, under a free licence, with the tests beside it.

Appendix: The Bundle's Keys

key	command
d / C-c C-d	firstpair-reader-describe-word
, . / C-c C-p C-c C-n	previous / next marked word
j k	next / previous source word, looked up
J K	next / previous significant word, looked up
RET	next source word looked up, or follow the link at point
[]	previous / next edition at point (Tr< Tr>)
v / C-c C-v / mouse-3	swap the edition at point
b / C-c C-b	add favourite editions (2nd)
B	keep only favourite editions (hold 2nd)
t T / C-c C-t	cycle dictionary and translation languages for- ward / backward (Lang)
= / C-c C-s	report the translations showing
r / C-c C-r	references; g / C-c C-g glossary; C-c C-f open a delivered file
C-c C-l	restore the standard layout; C-c C-o cycle the three windows
?	the key and tap summary
m (dictionary)	more / fewer senses; q close

Appendix: Reader Versions

version	what it brought
1.3	bundles identified by directory; two editions of one book registered together
1.4	dictionary languages per title; C-c C-t cycles them
1.9	touch: xterm-mouse-mode, header-line buttons, tap to look up
1.10–1.15	j/k word steps; focus sequences decoded; word arrows walk the source; one bar, buttons act on the tapped window
1.16	indirect Info subfiles; install-info copies them
1.17–1.19	compact dictionary, Italian headwords, RET navigation, lemma-true glosses, bundled reader beats a stale package
1.31–1.38	terminal menus with named commands, deferred feedback, header row, the persistent navigation contract with 1Unix
1.39	checkbox menus; ordered selections per language; header row hides and reorders; blocks permuted in place
1.41–1.43	2nd adds favourites, hold keeps only them; terminal wheel scrolling bound locally
1.45–1.46	first-load progress in the echo area; word controls right-aligned and doubled in width
1.47	region cache invalidated on Info's real file reads

Acknowledgments

Mark Liberman’s Spanish reader in Emacs, at Penn in the early 1990s, is the ancestor of everything here; the debt is thirty years old and gladly acknowledged. The authors of iSH built the impossible thing that made a phone into a Linux box, and hterm’s authors built the terminal that box draws on; 1Unix is a few hundred grateful lines on top of both. The English, Italian, and Russian Wiktionaries, extracted by Wiktextextract and published by Kaikki, are the dictionary; FreeDict and a small reviewed supplement fill its gaps. Longfellow, Cary, Norton, and Sibbald; Min, Petrov, Fedorov, Chuiko, and Minaev — and, in the study edition only, Lozinsky, Ilyushin, Palma, and James — are the voices beside the Italian.

The reader was written in sessions with Claude and Codex over the late summer of 2026, each taking turns at the same files; the mistakes recorded in this book are shared, and so is the credit. The bugs were found by reading on an iPhone in a terminal, one tercet at a time.