appligent
DOCUMENT SOLUTIONS

PageWright™

Developer Manual

Integrating, driving and extending the layout engine.

Appligent, Inc.
22 East Baltimore Avenue
Lansdowne, PA 19050 U.S.A.

(610) 284-4006
sales@appligent.com
https://www.appligent.com

Version 1.0
October 2026

© 2026 Appligent, Inc. All rights reserved. PageWright is a trademark of Appligent, Inc.

About this manual

This is the **PageWright™ Developer Manual**. PageWright is a PDF page-layout engine: you describe a document as a **JSON spec** - its pages, text, tables, images, charts, forms and structure - and the engine flows it onto pages and writes a tagged, standards-compliant PDF.

Every page of this manual was produced by the engine itself from a JSON spec (`sample/guide-developer-manual.json`), so it doubles as a worked example: the cover, the table of contents, the running heads and folios, the code boxes, the reference tables, and the index and glossary at the back are all ordinary engine features described in these pages.

> **Note**
>
> Conventions. Monospace text in a red tint marks a JSON key, value, or command - for example `"type": "paragraph"` or `pagewrightapp`. Shaded boxes are complete, valid snippets you can paste into a spec. **Bold** marks the first mention of a defined term; the Glossary collects them.

Contents

About this manual ii
Part I — Architecture 1
1. The two-pass flow model 1
2. Geometry & the render boundary 1
3. The text pipeline: bidi, shaping & /ToUnicode 2
Part II — The JSON Spec & Schema 5
4. Anatomy of a spec 5
5. Validating against the schema 6
Part III — Driving the Engine 7
6. The command line 7
7. The apDocLib server 9
8. How the applications are built on the engine 9
Part IV — Design-Tool Support 11
9. The layout map 11
10. Positioning, stylesheets & font enumeration 12
Part V — Tagged PDF / PDF-UA Internals 13
11. The structure tree 13
12. Conformance beyond the tree 14
Part VI — PDF/A & Print Production 16
13. PDF/A archival output 16
14. Print production 16
Part VII — Performance 18
15. Where the time goes 18
Part VIII — Extending the Engine 20
16. Anatomy of a feature 20
Appendices 21
A. Top-level key reference 21
B. Command reference 23
C. Dependencies 24
Glossary 25
Index 26

Part I — Architecture

1. The two-pass flow model

The engine's central idea is **two passes**. Many layout decisions depend on facts that are only known *after* layout - the total page count printed in a footer, the page a heading lands on for its table-of-contents entry, the page a threaded story is continued on. The engine resolves these by laying the document out more than once:

1. A **counting pass** runs the full layout against a metrics-only target that measures glyphs and boxes but draws nothing. It records where every heading, index term and frame landed.
2. Because the resolved numbers can themselves change the layout (a table of contents that grows by a line pushes everything down), the counting pass repeats as a **fixed-point loop** until the page numbers stop moving - usually two or three iterations.
3. A final **render pass** then draws the document once, pouring in the now-exact numbers and emitting the PDF.

Counting passes are cheap (no ink, no font embedding), so the loop converges fast. The table of contents, `{{pages}}` totals, the index and glossary, and article-threading jump lines all ride this one mechanism.

> Note
>
> Determinism falls out of this design: the same spec always produces the same bytes. The engine avoids wall-clock and random values; time tokens like `{{date}}` are sampled once per render and threaded through both passes so the counting and final layouts agree.

Why the loop converges

A naive worry is that resolving a page number changes the layout, which changes the page number, forever. It converges because each unknown reserves a **fixed amount of space** the moment its flag is set, independent of the digits it will eventually show: a `{{pages}}` footer is one line tall whether the total is 9 or 99; a TOC row is one line whether it points to page 3 or 30; a jump line reserves one line whether it reads "page 4" or "page 14". So the *geometry* stabilizes in the first pass or two; only the printed numerals refine afterwards, and they cannot move anything. In practice two or three counting passes suffice, and the engine stops as soon as a pass changes no page number.

> Note
>
> The same determinism discipline governs measurement, not only numbering. **Content-fit `auto` table columns** size each column from its content's own width - a fresh text measurement made during layout - yet must come out identical in the counting and final passes or pagination would desync.
>
> They do, because the counting pass measures through the very same font metrics the render pass uses; a column whose content cannot be measured (a list, a chart, a nested table) falls back to an even split. Any feature that measures during layout must obey this rule.

2. Geometry & the render boundary

Engine math is `double`, in PDF points, with a **top-left origin and y increasing downward** - the space the layout map and `position` use. The backend flips to PDF's bottom-left, y-up convention at emit time.

The engine never calls the PDF library directly. It speaks only to an abstract `IRenderTarget`, which has two implementations: `APDFLRenderTarget` (writes PDF via the Adobe PDF Library) and `MetricsOnlyTarget` (measures only - the counting pass). This boundary keeps the layout math unit-testable and leaves room for a non-APDFL backend.

Layer	Responsibility
Model	The document model: `LayoutDocument`, the block tree.
JSON	JSON spec -> model (rapidjson); tolerant parsing.
Engine	Flow, pagination, sections, the two-pass loop, running bands.
Text layout	Line breaking, alignment, widow/orphan, floats.
Block layout	Per-block `Measure`/`Place`: tables, charts, barcodes, frames.
APDFL backend	Content streams, fonts, tags, metadata.
Render boundary	`IRenderTarget`, the line between the engine and the backend.

Pagination is built on **block fragmentation**: a block's `Place()` returns the fragment that fit plus an optional *remainder* to continue on the next page or column. A paragraph splits between lines, a table between (or within) rows, a container between its children - and each remainder carries the tagging and style context so the continuation stays correct.

3. The text pipeline: bidi, shaping & /ToUnicode

Between the characters in the spec and the glyphs in the content stream sit four transforms, in this order:

Stage	What it does
Bidi resolution	UAX #9 over ICU's `ubidi`: paragraph level from the first strong character, then a split into level runs.
Shaping	GSUB lookups turn characters into the glyphs the face actually cuts - ligatures, Arabic joining forms, Indic conjuncts.
Positioning	`kern` / GPOS pair adjustment, applied to the glyphs shaping produced - not to the original characters.
Emit	Glyph ids into the content stream, plus an authored `/ToUnicode` CMap and `/ActualText` where the visual order was reversed.

> Caution
>
> **The invariant that holds the whole thing together: whatever the engine draws, it measured.** Both passes consume the same shaped, kerned glyph array. Measure a string of characters and draw a string of glyphs and justification drifts - a ligature is narrower than the letters it replaces, and a shaped Arabic word is not the sum of its isolated forms.
>
> Every bug in this family looks like bad spacing, so it is worth stating as a rule rather than an implementation detail.

Bidi: UAX #9 over ICU

The bidi pass is a thin wrapper over ICU's `ubidi` API. It resolves the paragraph level, splits the text into runs of uniform embedding level, and hands the layout code pieces it can lay out left-to-right. The **visual reorder happens at draw**, not at measure: line breaking works in logical order, and only when a line is committed are its pieces emitted in visual order.

A reversed piece carries **`/ActualText`** naming the logical string it stands for. That is what keeps copy, search and a screen reader reading the sentence the author wrote rather than the order the glyphs were painted in.

> **Note**
>
> [!] That `/ActualText` has a consequence for testing, and it is the single most important thing on this page: **text extraction cannot catch a reordering bug**. Ask a PDF extractor what a bidi page says and it reads back the `/ActualText` - the logical string - no matter what order the glyphs were actually drawn in.
>
> A reordering regression would extract perfectly. The oracle has to read the content stream.

Shaping and the fallback path

The engine parses the face's `GSUB` table, including the contextual lookup types (5 and 6) that Devanagari reordering needs. Arabic joining and the lam-alef ligature come out of the same machinery, and so do Indic clusters and reph.

When a face has **no usable GSUB**, Arabic falls back to the Unicode **presentation forms** block - the precomposed isolated/initial/medial/final codepoints. The text is readable, but it is a different mechanism with different limits: presentation forms have no joints, so `kashida` justification has nothing to elongate and silently does nothing. The gate distinguishes the two paths by signature rather than trusting either.

Why the engine authors its own /ToUnicode

A shaped run is drawn by **glyph id**, not by character code. Nothing in the font's own encoding maps those ids back to what the author typed - a ligature glyph is not `f` followed by `i`, and a Devanagari conjunct is not any single character. So the engine builds the `/ToUnicode` CMap itself, from the character-to-glyph correspondence shaping recorded on the way through.

> **Caution**
>
> Skip that and the document still *looks* perfect and is still tagged - but the text cannot be copied, searched, or read aloud, and the PDF/UA claim over it is false. This is the failure mode that ships silently: nothing on the page shows it, and only an extraction check finds it.

The oracles

None of the above is verified by looking at a page. Each stage is checked against an **independent implementation**, and each check carries a negative control that proves it can fail:

Oracle	What it proves
Bidi conformance	The Unicode Consortium's own **91,670-case** `BidiCharacterTest`, run against our resolver. The reference implementation, not our opinion of it.
Drawn order	Reads glyph codes **out of the content stream** and checks the drawn order - because `/ActualText` defeats extraction (see above).
GSUB parse	Our GSUB parse vs **fontTools**, per font, lookup by lookup.
Indic shaping	Our shaped output vs **HarfBuzz**, word by word, over eight Indic corpora - Devanagari, Bengali, Gurmukhi, Gujarati, Tamil, Odia, Telugu and Kannada, across 21 faces.
Shaping engaged	That shaping *happened* - ligature glyphs, alternates and presentation forms counted by signature, so a silently-unshaped run cannot pass as shaped.
Glyph to Unicode	That every glyph run maps back to the characters it stands for.

> **Note**
>
> The negative controls are the point. A corrupted glyph run must be caught; a blinded CMap must be caught; turning the features off must make the ligature signature disappear. A checker nobody has watched fail is a checker nobody should trust, which is why the regression gate reports each control by name.

One of these oracles was itself wrong for a while, which is worth knowing before you trust a green result. The drawn-order check used to union every `/ToUnicode` CMap in the file into one lookup, assuming a code meant the same character in every subset. It does not - subset encodings are per font, and one document embedding two RTL faces had 75 colliding codes. It produced false failures, and the same wrong decode could equally have **masked** a real reordering. It now decodes per font.

Part II — The JSON Spec & Schema

4. Anatomy of a spec

A spec is one JSON object. The keys most documents use - Appendix A lists all 56, generated from the schema:

Key	Type	Meaning
`title` / `author` / `subject` / `keywords` / `creator` / `producer`	string	Document metadata, written to both the `/Info` dictionary and the XMP packet. A `meta:{...}` object is also accepted, and is the only home of `rights`: a copyright or licence notice, written to XMP `dc:rights` alone (`/Info` has no rights entry).
`pageSetup`	object	Page geometry and furniture (size, margins, columns, header, footer, numbering, bleed, marks, watermark, template, ...).
`defaultStyle`	object	Base text style inherited by every block.
`styles`	object	Named stylesheet; entries may use `basedOn`.
`blocks`	array	The body as one implicit section.
`sections`	array	Explicit sections, each with its own `pageSetup` overrides + `blocks`.
`stories`	array/object	Free-standing block lists for threaded `frame` blocks.
`tagged` / `pdfua`	bool	Emit tagged PDF/UA.
`pdfa`	string/bool	Emit PDF/A (`"2b"`/`"2u"`/`"2a"`; `true` picks `2a` when tagged else `2u`).
`lang`	string	Document language (`/Lang`).
`colorMode`	string	`"rgb"` (default) or `"cmyk"` (convert all colors to DeviceCMYK).
`outputProfile`	string/object	The CMYK press's ICC profile to convert to and declare instead of SWOP (cmyk mode / PDF/X); `{profile, identifier, condition, info, registry}` names a registered condition.
`overprint`	bool	Spot inks overprint rather than knock out.
`hyphenate`	bool	Knuth-Liang en-US hyphenation.
`images`	object	Output policy for rasters: `maxPpi`, `format`, `jpegQuality`, `predictor`, `colorProfiles`; `minPpi` reports images placed below a resolution.
`viewer`	object	Initial view: `layout`, `mode`, `fit`, `page`.

Metadata is written twice for interoperability: Unicode-safe into the **`/Info`** dictionary (`PDDocSetInfoAsASText`) and into an **XMP** packet (`PDDocSetXAPMetadata`), so a reader that trusts either source sees the same values. Two fields have defaults you never have to set: **`/Creator`** is `PageWright` (the tool that laid the page out), and **`/Producer`** is `Appligent PageWright` (the library that wrote the file) - the latter in both `/Info` and XMP `pdf:Producer`. Override the producer when you ship the engine inside your own application:

```
{
  "title": "Quarterly Report",
  "author": "Finance",
  "producer": "Acme Report Server 4.2",
  "blocks": [
    {
      "type": "paragraph",
      "text": "..."
    }
  ]
}
```

> **Tip**
>
> Confirm what shipped with `pdfinfo out.pdf` (the `/Info` view) and `pdfinfo -meta out.pdf` (the XMP packet); both report the same `Producer`.

5. Validating against the schema

The complete grammar is published as a JSON Schema (draft 2020-12) at `aplayout.schema.json` - every block type, style key, color form, page-setup key, chart and form field. The engine checks every spec against it on the way in, whichever way the spec arrives - a single render, a batch or the server:

Mode	A spec that does not match the schema
default	Each problem is printed - `aplayout: <message> (at /pointer)`, up to 20, then a count of the rest - and the render carries on. The exit status is unaffected.
`-strictschema`	Refused: `spec error: the spec does not match aplayout.schema.json`, no PDF, exit 1.
`-noschemacheck`	The check is skipped.
`-checkschema <spec>...`	Check only, render nothing: the problems are printed, each file is named with its verdict, then `K of N did not match`. Exit 0 when all match, 1 when any does not (a file that is not JSON counts), 2 for a usage error.

The engine finds the schema at `$APLAYOUT_SCHEMA` when that is set, then in the Appligent home (`-aphome`), then in `/usr/local/appligent/aplayout/` and `/Applications/Appligent/`; when it finds none it says the spec was not checked. Point an editor at the same file for completion and inline validation as you type. A CI job without the engine can use any JSON Schema validator, such as `check-jsonschema --schemafile aplayout.schema.json mydoc.json`. The schema is the contract; this manual is the explanation.

Part III — Driving the Engine

6. The command line

`pagewrightapp` renders one spec to one PDF:

```
pagewrightapp spec.json out.pdf            # render
pagewrightapp spec.json out.pdf -map out.map.json   # + a layout map
pagewrightapp -batch jobs.txt               # many documents, ONE process
pagewrightapp -batch jobs.txt -jobs auto    # ... across several worker processes
pagewrightapp -fonts                        # list embeddable system fonts (JSON)
```

Flag	Effect
`-map <path>`	Also write the layout map (*The layout map*) beside the PDF.
`-maponly`	Lay out and write only that map - no PDF. Measurement without the save.
`-impose <path>`	Also write a PRINT copy: the pages of the PDF two to a sheet side in saddle-stitch order, to print both sides, fold and staple. Laid out as the spec's `impose` says, or two pages side by side; untagged, and its /Subject says how to print it. The PDF itself is written as always.
`-batch <list>`	Render every `<spec><TAB><out.pdf>` line of a job list in one process. `#` comments and blank lines are skipped.
`-jobs N/auto`	Spread a `-batch` list over N worker processes (default 1). Documents are independent, so this scales until the workers contend for memory bandwidth and each re-pays the font warm-up. Peak memory multiplies by N and the per-document lines interleave; the summary and exit status do not change.
`-data <file>`	Bind a JSON/CSV data file and render once per record; `-merge` writes one combined PDF instead of numbered files.
`-condition "T=on"`	Override condition tags at build time - one source, many editions.
`-progress`	Report progress as pages are laid out.
`-fonts`	Print every system font APDFL can see, as JSON, each with a `usable` flag from the renderer's own resolver, and exit.
`-auditfonts <spec>`	Font preflight: per family, whether the finished file carries it, which file answered, characters set, characters it cannot draw. Writes no PDF. **Exit 2 = findings to review**, so it drops into a build script.
`-auditreport`	Write that same record beside each output as `<out.pdf>.fontaudit.json`. Composes with `-batch`, so a whole corpus is audited by the render that was happening anyway.
`-allowunembeddedfonts`	Downgrade the unembedded-font refusal to a warning and keep the file. For diagnosing such a document, not for shipping one.
`-checkschema <spec>...`	Check specs against the schema and render nothing; exit 0 when all match, 1 when any does not (*Validating against the schema*).
`-strictschema` / `-noschemacheck`	Refuse a spec that does not match the schema (no PDF, exit 1) / skip the check. By default problems are reported and the render goes on.
`-fontdir <dir>`	Add a font folder, searched before the installed fonts. Repeatable, merged with the spec's `fontDirs`, and accepted by `-batch` and `-fonts`. An installed font of the same name still wins.
`-aphome <dir>`	The Appligent home: where the APDFL resources, the font search, the schema and (on macOS and Linux) the licence are found. Default: `$APPLIGENT_HOME`, else `/Applications/Appligent`, else `/usr/local/appligent`.
`-license [-product <code>]`	First on the line: print the licence status as one JSON line - the product, the licence number (masked), whether it is licensed - and exit, rendering nothing.
`-v`	First on the line: print the version as one line - `PageWright 1.0 (<build date>), APDFL21.0.0a` - and exit, rendering nothing.
`-product <code>`	The Appligent product whose licence covers the run - `PGWR`, PageWright, by default. A named product that is not licensed on the machine marks every page as an evaluation. Appligent's own applications pass it, with a `-ticket` where theirs needs one.

A relative asset `src` resolves against the **spec file's** directory, so a spec travels with its images and placed PDFs. The output path is resolved against the **working** directory, like any other command-line tool.

> Tip
>
> **Rendering many documents: one process, then several.** APDFL builds its system-font catalog the first time a document embeds a font - about 560 ms, paid once per *process*, not per document. A one-shot render therefore spends most of its time initializing, and a shell loop pays that for every file.
>
> `-batch` (or the apDocLib server, or `-data` for a data merge) pays it once. Then `-jobs auto` spreads the list over worker processes, because one document never depends on another. Rendering this engine's sample corpus: about 126 s as one command per document, about 10 s as a single `-batch` run, and about 3 s with `-jobs auto` on an 18-core machine.
>
> Worker count is not free - each one re-pays the font warm-up and they share memory bandwidth, so the curve flattens and eventually reverses.

7. The apDocLib server

For applications that render many documents, the engine ships as an `apDocLib` server command, `LayoutDocument`, driven over stdin exactly as a host application (for example a Xojo app via `Lib_Shell`) drives it. The library and fonts initialize once, so each subsequent render costs only its own layout time:

```
printf 'LayoutDocument -infile spec.json -outfile out.pdf\nexit\n' \
  | apdoclibapp -start
```

Parameter	Meaning
`-infile <spec.json>`	The input document spec.
`-outfile <out.pdf>`	The output PDF (absolute path).
`-mapfile <out.map.json>`	Optional layout map (consumed by that command only).

> Tip
>
> The server is the production path. The first request pays the one-time ~540 ms font warm-up; every request after it renders a typical tagged document in roughly 8-10 ms. See Part VII.

8. How the applications are built on the engine

The engine is **not distributed as a standalone library**. It is delivered as the command-line tool, the apDocLib server and the PageWright Studio designer; an application that needs to render documents drives the server, which is the supported integration path and the one Part VII measures.

This chapter is here so the internals are legible, not as an interface to build against. Appligent's own applications link `LayoutEngine` against an `APDFLRenderTarget`: the model is built from JSON (or constructed directly), and `Render` runs the two-pass flow, optionally returning a layout map.

```
LayoutDocument doc;
LoadDocumentFromFile("spec.json", doc, err);

APDFLRenderTarget target(pdfDoc, doc.info);
LayoutMap map;
LayoutEngine engine;
engine.Render(doc, target, &map);               // counting + render
engine.WriteLayoutMap(map, "out.map.json", err);
```

Server integration is mechanical: declare `Do_LayoutDocument`, add a `LayoutDocument` case to the command dispatcher, parse the spec into the model, run the engine against an `APDFLRenderTarget`, and save.

Part IV — Design-Tool Support

9. The layout map

Four features exist so an interactive editor can sit on top of the engine. The first is the **layout map** (`-map`): alongside the PDF the engine writes where everything landed, collected on the final pass. Each block is keyed by a **JSON Pointer** into the source spec, so a tool hit-tests a click to a path, mutates that node, and re-renders.

```
{
  "version": 1,
  "units": "pt",
  "origin": "top-left",
  "pages": [
    {
      "width": 612,
      "height": 792,
      "label": "1",
      "content": {
        "x": 64,
        "y": 64,
        "w": 484,
        "h": 664
      }
    }
  ],
  "blocks": [
    {
      "path": "/sections/1/blocks/3",
      "type": "table",
      "id": "ledger",
      "fragments": [
        {
          "page": 2,
          "x": 64,
          "y": 64,
          "w": 484,
          "h": 503
        },
        {
          "page": 3,
          "x": 64,
          "y": 64,
          "w": 484,
          "h": 96
        }
      ]
    }
  ]
}
```

Field	Meaning
`pages[]`	One per physical page: `width`, `height`, folio `label`, owning `section`, and the body `content` box.
`blocks[].path`	A JSON Pointer to the source node (`/sections/1/blocks/3`, `/blocks/2/items/1`, `/stories/0/blocks/2`) - the GUI's edit handle.
`blocks[].type` / `id`	The block's type, and its `id` when it declares one.
`blocks[].fragments[]`	The flow band(s) it consumed - `{page, x, y, w, h}` - one per page/column slot (a split block has several).
`blocks[].visual[]`	Precise drawn rectangles where the flow band is not the truth: the image / chart / placed-PDF / barcode box, and each form-widget box.

Coordinates are points with a **top-left origin**, matching a raster of the page after scaling by `dpi / 72`. The map records *spec* blocks only - engine- synthesized content (running bands, footnote bodies, repeated table headers, watermarks) has no pointer and is deliberately absent. Ancestry is derivable from the path string, so a tool resolves a click to the deepest entry and walks outward. Engine API: `LayoutEngine::Render(doc, target, &map)` then `WriteLayoutMap(map, path, err)`.

10. Positioning, stylesheets & font enumeration

The other three editor features:

- **Absolute positioning** (`position` on any block) - direct manipulation and form overlays onto existing PDFs. Map `visual` rect equals `position` rect, so a drag writes straight back to the spec.
- **Named stylesheets** (`styles` + `basedOn`) - the "edit once, document follows" primitive: keep styles in the table, reference by name, and one edit restyles every use on the next render.
- **Font enumeration** (`pagewrightapp -fonts`) - a picker offers only fonts the engine can actually resolve and embed.

The companion *PageWright Studio* application is a full document/form designer built on exactly these four hooks: render with `-map`, raster the page, drag the `visual` rectangles, write new `position` values back through the JSON Pointers, re-render (the server mode renders in ~10 ms).

Part V — Tagged PDF / PDF-UA Internals

11. The structure tree

With `"tagged": true` the engine builds a logical **structure tree** and marks the content stream so each span belongs to a structure element. The block -> tag mapping:

Block	Structure element
heading (level 1-6)	`H1`-`H6`. Each also opens a nested `Sect` under the default `sectionTags: auto`; see below
paragraph	`P` (with `/Lang` if the block sets one)
image / chart / placed PDF	`Figure` with `/Alt`, plus `Caption`
table	`Table` › `THead`/`TBody` › `TR` › `TH`/`TD` (header `/Scope`) - or, with `rowGroups`, ONE `Table` with no `THead`/`TBody`; see below
callout / admonition / note	`Aside` (role-mapped to `Div` before PDF 2.0)
container	nothing, unless it declares a `role` - then `Aside`, `Sect`, `Div` or `BlockQuote`
list	`L` › `LI` › `Lbl` + `LBody`
toc	`TOC` › `Caption` + `TOCI`
link run	`Link` wrapping the text + an `OBJR` to the annotation
form field	`Form` holding only the widget's `OBJR` (one per `Form`)
footnote	`Note` with an `/ID`
inline `lang` run	`Span` carrying `/Lang`
rules, borders, dot leaders, heads/footers, marks	`Artifact` (out of reading order)

The engine assigns its own **MCID**s (page-scoped, reset per page) so the raw operator stream and the embedded-text PDE containers share one numbering, then builds the `StructTreeRoot`, the `/K` child arrays in reading order, and a **ParentTree** mapping page + annotation entries back to elements. The `StructTreeRoot` also gets a `/RoleMap` when a non-standard type is present, and under PDF/UA-2 a `/Namespaces` array - with `/NS` on the `Document` root and on every `Aside`, a type that exists only in PDF 2.0. The catalog gets `/MarkInfo`, `/Lang`, `/ViewerPreferences/DisplayDocTitle`, and the XMP gets `pdfuaid:part`.

Three things about that mapping are the author's choice, not the engine's. `sectionTags` decides whether a heading opens a nested `Sect` at all (`auto` / `pages` / `none`); a `container`'s `role` decides whether the box is a structure element or nothing; and a table with `rowGroups` produces a different tree entirely - one `Table` across every page it runs to, group headings lifted into rowspanned `TH` cells in synthetic leading columns, and `/Headers` derived on every cell. All of it is PDF/UA-1 clean either way; what differs is what a screen reader walks through. The Users Manual, Part VIII covers the authoring side.

Two backend mechanisms make the last of those possible, and they are the only places the structure tree is allowed to diverge from the order content is drawn in. **`OpenStructDeferred` / `AdoptStruct`** open an element with no parent and let a later row adopt it as the head of its children - that is how a heading drawn in its own full-width row becomes the first cell of the row beneath it without the ink moving. **`HoldStructOpen` / `ReleaseHeldStruct`** keep one element open across a page break, which is what makes a single `Table` legal over three pages: an element's `/K` holds marked-content references that each name their own `/Pg`, so one element

may reference content on any number of pages. A deferred element that is never adopted refuses the document rather than being written dangling.

> **Note**
>
> Two content paths feed one tree. Base-14 text and graphics inject `/Tag <</MCID n>> BDC ... EMC` straight into the operator stream; embedded text is drawn after the graphics as PDE text and wrapped in marked-content containers, so structure survives APDFL's PDE round-trip. Both share the per-page MCID counter.

The structure-event API

The layout engine emits structure through `IRenderTarget` hooks that are no-ops on the metrics/counting targets - so counting passes build no tags, and only the final pass produces the tree. Five carry ordinary tagging:

```
bool Tagging() const;                    // gate: skip tagging when off
void BeginStruct(const StructAttr&);     // open P / H1 / Figure / Table / Aside / ...
void EndStruct();
void BeginArtifact(ArtifactKind);        // Pagination | Layout | Page
void EndArtifact();
```

Four more exist for the one case where the tree must **diverge from the order content is drawn in** - a hierarchical table. They are deliberately generic rather than table-specific, and `SuspendStructure` / `ResumeStructure` complete the set by parking the open scope while a footnote body or running band is drawn mid-flow:

```
int  OpenStructDeferred(const StructAttr&); // an element with NO parent; returns a
  handle
void AdoptStruct(int handle);               // make it a child of the CURRENTLY open
  element
int  HoldStructOpen();                      // the innermost element outlives its
  caller's scope
void ReleaseHeldStruct(int handle);         // ...which is how ONE <Table> spans a page
  break
```

`StructAttr` carries eleven fields - `{type, alt, actualText, lang, mathML, scope, listNumbering, colSpan, rowSpan, id, headers}` - and the last four are what make PDF/UA table association expressible: `id` names a cell, `headers` lists the cells that govern it. RAII guards (`TagScope` / `ArtifactScope`) keep the open/close balanced.

Structure elements are created **per `Place` call** by default, so a paragraph split across pages becomes two valid `<P>` and a split table re-emits a complete `<Table>` per fragment with its header rows re-tagged. That is this engine's habit, not a rule of the format: an element's `/K` holds marked-content *references* that each name their own `/Pg`, so one element may span pages. A `rowGroups` table does exactly that - one `<Table>` for the whole run via `HoldStructOpen` / `ReleaseHeldStruct`, with the repeated column band on later pages artifacted - and a split call-out re-opens its `<Aside>` on each continuation page.

Adding virtuals shifts the `IRenderTarget` vtable, so every translation unit must be rebuilt.

12. Conformance beyond the tree

Four things make a tagged file actually pass PDF/UA-1, each learned from a real veraPDF failure:

1. **All fonts embedded** (7.21.4.1) - Base-14 cannot embed, so a tagged doc with no `embedFont` auto-embeds a Helvetica subset.

2. **No inconsistent `/CIDSet`** (7.21.4.2) - APDFL's subsetter writes one that veraPDF rejects; `StripCIDSets()` removes it (and recurses into imported Form-XObject resources for placed PDFs).
3. **Links fully tagged** (7.18) - the `Link` element, an `OBJR` to the annotation, `/Contents`, and a `/StructParent` round-tripped through the ParentTree.
4. **A `/RoleMap` for any non-standard type** - `<Aside>` does not exist before PDF 2.0, so emitting one without mapping it fails outright. Whenever an `Aside` is present and the target is not PDF/UA-2, the engine writes `/RoleMap << /Aside /Div >>` on the `StructTreeRoot`. Under PDF/UA-2 the type is standard and is namespaced instead, so no map is written - remapping a standard type there would describe it as something it is not.

The authoritative check is `verapdf --flavour ua1 out.pdf`; every tagged sample passes with zero failures.

Form widgets and per-element language

Two more cases the engine handles so authors do not have to. A form widget is wrapped in its **own** `<Form>` element that holds *only* the widget's `OBJR` (PDF/UA clause 7.18.4) - the visible label is a sibling `<P>`, and a radio group becomes several `<Form>`s, one per option, each with a `/StructParent` and a `/TU` accessible name. A pre-filled text or choice value is stored as a UTF-16BE string so its appearance font embeds, satisfying 7.21.4.1.

Per-element language threads through too: a block `lang` sets `/Lang` on its element (and on every fragment of a page-split paragraph), while a run `lang` wraps just that run in a `<Span>` with `/Lang`, grouped as one Span per contiguous same-language run - not per word.

> **Caution**
>
> A document-level `/Lang` alone keeps veraPDF `ua1` clean; per-element `/Lang` is a correctness win for assistive technology, not a conformance fix. But alt text on figures and a document title genuinely are required - the engine supplies a title if you omit one, but it cannot invent alt text for an image.

Part VI — PDF/A & Print Production

13. PDF/A archival output

`pdfa` selects an ISO 19005-2 level. At `FinishDocument` the engine emits the conformance machinery: an `/OutputIntents` entry (`/S /GTS_PDFA1`) whose `/DestOutputProfile` is a **bundled ICC profile** - sRGB by default, or U.S. Web Coated SWOP v2 (or the document's `outputProfile`) when `colorMode` is `cmyk` - so the file is self-contained; a `pdfaid` XMP packet with the part and conformance letter; and enforced font embedding. Level `2a` additionally turns tagging on and emits a `pdfuaid` extension-schema description so the one file is valid PDF/A-2a **and** PDF/UA.

Level	Adds	Engine provides
`2b`	Reproducible rendering	Embedded fonts + OutputIntent + pdfaid XMP + trailer `/ID`
`2u`	Unicode for all text	ToUnicode CMaps (always written)
`2a`	Tagged structure + reading order	Tagging + the pdfuaid extension schema

The `/OutputIntents` entry the engine writes onto the catalog, with the ICC profile embedded as a compressed stream so the file is self-contained:

```
/OutputIntents [ <<
  /Type /OutputIntent  /S /GTS_PDFA1
  /OutputConditionIdentifier (sRGB IEC61966-2.1)
  /Info (sRGB IEC61966-2.1)
  /DestOutputProfile <ICC stream, /N 3, FlateDecode>
>> ]
```

The sRGB and SWOP profile bytes are bundled in the binary, so there is no runtime dependency on an installed profile. `/S /GTS_PDFA1` is the subtype for **every** PDF/A part. The XMP gains `pdfaid:part` + `pdfaid:conformance`; a `2a` file additionally carries a `pdfuaid` extension-schema description (PDF/A only permits predefined or PDF/A-extension schemas, and `pdfuaid` is neither), without which `--flavour 2a` fails clause 6.6.2.3.1.

14. Print production

Print furniture lives in `pageSetup`. `bleed` (a scalar or per-edge insets) grows the MediaBox and sets `TrimBox`/`BleedBox` for imposition while content stays in trim coordinates. `marks` strokes crop marks and registration targets - in a real `/Separation /All` ink so they print on every plate - outside the bleed. `background`, `colorBars` and `slug` complete the slug area. All of it is drawn as a **Page artifact**, so a tagged PDF/A-2a file with bleed and marks still passes both `--flavour 2a` and `--flavour ua1`.

Box (Letter, 9 pt bleed, marks)	Value
`/MediaBox`	`[-33 -33 645 825]` (trim + bleed + marks area)
`/BleedBox`	`[-9 -9 621 801]` (trim + bleed)
`/TrimBox`	`[0 0 612 792]` (the finished page)

> **Caution**
>
> Top-level `overprint` wraps spot fills in a `/OP /op true` ExtGState. Because APDFL hoists graphics-state changes outside marked content, pair `overprint` with **untagged** output (PDF/A-2b is fine). Embedded spot and 100% K text overprint via a PDE-level state that stays inside the tag, so they are safe in tagged files.

Part VII — Performance

15. Where the time goes

The engine is fast; the one large cost is **font embedding**, and it is paid **once per process**, not per document:

Document class	Server (warm)	CLI (cold)
Base-14, untagged	2-12 ms	2-12 ms + ~48 ms init
Tagged / embedded font	~8-10 ms	540-850 ms (one-time font warm-up)

Tagging itself is nearly free (about 7 ms on a 27-page document); but PDF/UA requires embedded fonts, and the first embed triggers a ~540 ms system-font load that APDFL then caches process-globally. On the long-running server that cost is amortized to near zero - which is the whole reason to prefer the server for batches. Body layout runs ~2 ms/page (Base-14); the per-page Flate compression of the content stream is the next-largest cross-cutting cost. Building at `-O2` roughly halves the pure-layout path.

Feature isolation

Cost of turning on one feature at a time, on a 400-paragraph (~27-page) document (warm minimum, milliseconds):

Variant	Δ vs base	Takeaway
untagged Base-14	- (104 ms)	Baseline.
+ 2 / 3 columns	+2 / +3	Essentially free.
+ hyphenation	+4	Essentially free.
+ tagging (structure tree only)	+7	Tagging itself is nearly free.
+ **font embedding**	**+685**	The dominant cost (one-time per process).
+ auto-TOC & its counting pass	+96	Modest, includes the extra TOC page.

Scaling

Axis	Base-14	Tagged
Body text	~2.1 ms/page (470 pg/s)	~7.9 ms/page + ~600 ms font (once)
Table rows (5 cols)	~0.8 ms/row (linear)	~4.3 ms/row (super-linear)
Empty form field	-	~0.1 ms each (flat: 5 vs 40 fields equal)

Everything is linear except **tagged tables** of hundreds of rows, where re-tagging repeated header rows across many page fragments compounds - fine for ordinary documents, a consideration for data-table-heavy output.

CLI versus server, measured

Successive `LayoutDocument` requests to one warm server process (the `embed` doc - four embedded font variants):

Requests	Total	Marginal / request
1	597 ms	- (pays the ~540 ms warm-up)
8	651 ms	7.7 ms
16	749 ms	10.1 ms

> **Tip**
>
> The lesson: the server amortizes the font warm-up to ~8-10 ms per request. For high-volume work use the server (or a warm CLI process pool); the bare CLI pays the ~540 ms warm-up on every invocation that embeds a font.

For raster-heavy documents, the top-level `images` policy trims file size: `maxPpi` downsamples images placed above a resolution, `format` chooses pass-through JPEG vs lossless Flate vs re-encode, `jpegQuality` tunes re-encode quality, and `predictor` applies PNG row predictors to lossless images. `minPpi` changes nothing in the file: it reports, with the block's pointer, any image block placed below that effective resolution - a preflight for print.

```
{
  "images": {
    "maxPpi": 200,
    "format": "auto",
    "jpegQuality": 80,
    "predictor": true
  }
}
```

Part VIII — Extending the Engine

16. Anatomy of a feature

A new capability touches the same five layers, in order. The article-threading feature (`frame` blocks + `stories`) is a representative case study:

1. **Model** - a `FrameBlock` type and a `Story` / `stories` container on `LayoutDocument`.
2. **JSON** - parse the new block and the top-level `stories` into the model.
3. **Layout** - `FrameBlock::Place` runs a column loop that pulls from a resumable story cursor and threads the overflow forward.
4. **Engine** - own the per-document story pool, reset it each sweep, and resolve the jump-line page numbers in the fixed-point loop.
5. **Render** - usually nothing new: frames reuse the existing absolute-placement and cross-reference machinery, so no backend change was needed.

Most features follow this shape; the render layer changes only when a genuinely new drawing primitive is required (as `FillPolygon` was for pie and radar charts).

> Tip
>
> Build the layout core first against `MetricsOnlyTarget` - the engine backend boundary lets you unit-test pagination math with no PDF at all, then wire the backend once the geometry is right.

The Place / Measure contract

Every block implements two methods. `Measure` reports the height a block wants at a given width (used by tables, floats and keep-together decisions). `Place` draws as much as fits in the available band and returns a **remainder** - the part that did not fit - or none. The remainder is itself a block (often a lightweight `PreflowedText` / `WrappedFlow` carrying pre-broken lines), so the engine simply places it again at the top of the next column or page. This single contract is what makes pagination, column flow, and threaded frames all fall out of the same loop.

A new drawing primitive - when a feature truly needs one - is added to `IRenderTarget` and implemented on **both** targets (a real draw on `APDFLRenderTarget`, a no-op on the metrics target). `FillPolygon`, added for pie wedges and radar fills, is the model: a handful of lines on each target, emitted inside the chart's `Figure` so it stays PDF/UA-clean. Adding a virtual shifts the vtable, so rebuild every translation unit.

Appendices

A. Top-level key reference

Every top-level spec key, with where it is covered:

Every top-level spec key. This table is GENERATED from `aplayout.schema.json` at build time - the hand-maintained version of it named 23 of the 56 keys and called itself complete, which is the failure mode a generated table cannot have:

Key	Type	What it is
`attachments`	array	Associated files embedded as PDF/A-3 attachments (catalog /AF + /Names /EmbeddedFiles).
`author`	string	/Info Author + XMP dc:creator.
`baselineGrid`	number	Baseline grid pitch in points (0 = off): text-line baselines snap to a page-absolute grid (measured from the content top) so columns and facing pages align.
`binding`	enum	Which edge the book is bound on.
`blocks`	array	The document body as a single implicit section.
`canvas`	boolean	Read-only echo of canvas mode, which is turned on by the presence of "pages".
`changeBars`	object	Revision / change bars: the appearance of the vertical bar drawn in the page margin beside any content marked changed (a block's changed:true, or a block carrying a condition tag declared...
`chapters`	array	Native book model: assemble N chapter spec files into ONE PDF.
`colorMode`	enum	"cmyk" renders the whole document in DeviceCMYK (RGB inputs are converted, so the file carries no DeviceRGB) and a PDF/A document then gets a CMYK OutputIntent (the bundled U.S.
`conditionIndicators`	boolean	Conditional text author view: when true, conditional content is NOT dropped - every conditional block/run is tinted in its tag's color and the hidden ones are struck through, so an editor...
`conditions`	object	Conditional text (FrameMaker condition tags): maps a condition-tag name to its state - a bare boolean (true = shown), or an object { show, color }.
`creator`	string	/Info Creator + XMP xmp:CreatorTool (default "PageWright").
`data`	object / array	Data-driven binding context: a record (object) whose fields are substituted into every {{path}} token in the document before layout, e.g.
`dataFile`	string	Path (relative to the spec) to an external record used as the data binding context when no inline data is given.
`declarationGate`	enum	How the WCAG gate reacts when this document contradicts a WCAG level it declares in `declarations`.
`declarations`	array	PDF Declarations (PDF Association): machine-readable claims, written to the XMP pdfd:declarations bag, that this document conforms to a standard PDF has no identifier of its own for - WCA...
`defaultStyle`	object	Document-wide default style - the base every block inherits before its own style applies.
`fallbackFont`	string	CJK fallback font family for the engine's auto-generated blocks (toc / lof / glossary / index).

Key	Type	What it is
`features`	array	The document-wide OpenType feature set - what every run is set with unless its own style names a different one.
`fontDirs`	array	Extra font FOLDERS this document may name faces from, searched before the installed ones, so a spec can travel with a face that is not installed on the rendering machine.
`hangingPunctuation`	boolean	CJK hanging punctuation (ぶら下げ): a trailing ideographic comma/full stop (、。) or its full-width Latin form (，．) hangs past the line edge into the margin instead of wrapping to the next line.
`hyphenate`	boolean	Hyphenate Latin words at line ends (Knuth/Liang patterns; en, fr, de and es are embedded).
`images`	object	Output policy for placed raster images, and a low-resolution report (minPpi).
`impose`	object	How a print copy of this document is laid out on sheets: a saddle-stitched booklet, two pages to a sheet side, for sheets printed on both sides, folded once and stapled on the fold.
`kashida`	boolean	Kashida (tatweel) justification for Arabic: a JUSTIFIED Arabic line spends its slack elongating the connections between letters - inserting the face's own tatweel glyph (U+0640) at the jo...
`kerning`	boolean	Apply pair kerning: tightens pairs such as AV, To, Wa, F.
`keywords`	string	/Info Keywords + XMP pdf:Keywords.
`lang`	string	Document language, BCP-47 (e.g.
`ligatures`	boolean	Apply the embedded face's OpenType standard ligatures (the `liga` feature): fi, fl, ff, ffi and ffl are drawn as the single glyph the type designer cut for them instead of as colliding le...
`linkStyle`	one of	Make links LOOK clickable.
`master`	string	The document-wide DEFAULT master page (a key of "masters").
`masters`	object	Master pages: named, reusable page templates.
`meta`	object	Optional metadata object; its fields override the top-level ones.
`namedDestinations`	boolean	Publish every block "id" as a PDF named destination (catalog /Names /Dests), so `guide.pdf#nameddest=chapter3` opens at that block and other tools can deep-link into the document.
`notes`	object	Footnote / endnote streams: the marker glyph style and, for footnotes, the counter-restart scope.
`numbering`	object	Automatic content numbering.
`opticalMargins`	boolean	Latin optical margins: trailing punctuation (.
`optimalBreaks`	boolean	Knuth-Plass optimal line breaking for justified paragraphs: choose breakpoints that minimize the total badness across the whole paragraph (more even spacing, fewer rivers) instead of the...
`outputProfile`	one of	The CMYK press this document is printed on, as an ICC output profile file - in place of the built-in U.S.
`overprint`	boolean	Spot-ink (separation) fills and strokes overprint instead of knocking out (/OP /op true, /OPM 1).

Key	Type	What it is
pageSetup	object	Page geometry and print setup for the document: size, orientation, margins, columns, running heads and the print-production extras.
pages	array	CANVAS MODE: the document is a stack of DESIGNED pages rather than a flow.
palette	object	Document color palette.
pdfa	one of	PDF/A archival conformance: "2b"|"2u"|"2a" (PDF/A-2), "3b"|"3u"|"3a" (PDF/A-3, which also permits embedded files), or "4"|"4e"|"4f" (PDF/A-4, the PDF 2.0 revision; "4f" permits embedded f...
pdfua	one of	PDF/UA target.
pdfx	one of	PDF/X print-production conformance.
producer	string	/Info Producer + XMP pdf:Producer (default "Appligent PageWright").
recordIndex	integer	Which record of "records" this document binds (1-based).
records	string / array	A record SET for mail merge: a path (relative to the spec) to a JSON array or a CSV whose header row names the fields, or an inline array of record objects.
renderingIntent	object	How colors outside the destination's gamut are mapped, for the whole document.
restart	boolean	Book model: restart page numbering at 1 for every chapter (per-chapter folios).
sectionTags	enum	How much <Sect> GROUPING the structure tree carries.
sections	array	Explicit sections.
security	object	Standard PDF security (encryption + permissions) applied via aplib APEncrypt.
stories	one of	Free-standing block lists poured into the thread of "frame" blocks that reference them by id (article threading).
styles	object	Named stylesheet: each entry is a style object (optionally "basedOn" another entry).
subject	string	/Info Subject + XMP dc:description.
tagged	boolean	Emit a tagged PDF (logical structure tree).
title	string	Document title: /Info Title, XMP dc:title, and the {{title}} variable.
variables	object	Variable catalog: named values referenced with {{var.name}} anywhere the data binding reaches (text runs, links, image/pdf src, running-band templates, metadata).
viewer	object	Initial view: catalog /PageLayout, /PageMode and the /OpenAction destination.

B. Command reference

CLI - `pagewrightapp`:

```
pagewrightapp <spec.json> <out.pdf> [-map <out.map.json>] [-impose <booklet.pdf>]
pagewrightapp -batch <jobs.txt> [-jobs N|auto]
pagewrightapp -checkschema <spec.json> [<spec.json> ...]
pagewrightapp -fonts [-fontdir <dir>]  # JSON list of embeddable fonts
pagewrightapp -license               # licence status as JSON
```

Server - the `LayoutDocument` command on `apdoclibapp`:

```
LayoutDocument -infile <spec.json> -outfile <out.pdf> [-mapfile <map.json>]
```

Validate - against the published schema:

```
pagewrightapp -checkschema <spec.json>
check-jsonschema --schemafile aplayout.schema.json <spec.json>   # without the engine
```

Verify output - accessibility and archival conformance:

```
verapdf --flavour ua1 <out.pdf>     # PDF/UA-1
verapdf --flavour 2a  <out.pdf>     # PDF/A-2a
qpdf --check <out.pdf>              # structural check
```

C. Dependencies

The engine's dependencies, and what each is used for:

Dependency	Used for
Adobe PDF Library (APDFL) 21.0.0	PDF production, fonts, content streams
rapidjson	Parsing the JSON spec
zint	Barcode symbologies (vector output)
libpng / libjpeg / libtiff / libwebp	Decoding raster image formats
ImageIO + CoreGraphics (macOS only)	Decoding HEIC / HEIF / AVIF
bundled sRGB + SWOP ICC profiles	PDF/A OutputIntents (self-contained)

Glossary

Base-14 The fourteen fonts every PDF reader provides built in (Helvetica, Times, Courier families, Symbol, ZapfDingbats); usable with no embedding.

Bleed Extra printed area past the trim edge so ink runs to the edge after cutting; set with pageSetup.bleed.

Block One unit of content in a spec (paragraph, heading, list, table, image, chart, …), identified by its type.

Counting pass A metrics-only layout pass that draws nothing; it measures the document so forward references (TOC, totals, index) resolve before any ink is committed.

Embedded font A font subset stored inside the PDF so it renders identically anywhere; required for tagged and PDF/A output.

Form XObject A reusable PDF content stream; the engine imports an external PDF page or a letterhead as one, vector and unrasterized.

Frame An absolutely-positioned box that pours a story into its columns; frames referencing one story form a thread (article threading).

Layout map The -map sidecar JSON: for every spec block, the page and rectangles where it landed, keyed by a JSON Pointer into the source spec.

MCID Marked-content identifier: a per-page integer linking a span of page content to its element in the structure tree via the parent tree.

OutputIntent A PDF/A color-characterization dictionary naming the target color space via an embedded ICC profile (sRGB or CMYK).

PDF/A ISO 19005 archival PDF: self-contained and reproducible. Levels 2b, 2u, 2a in this engine.

PDF/UA ISO 14289 universal-accessibility PDF: a tagged logical structure tree with reading order and alternate text.

Point The PDF unit of length, 1/72 inch. All coordinates and sizes in a spec are points unless written as a percentage.

Render target The IRenderTarget interface between the layout engine and a backend; APDFLRenderTarget writes PDF, MetricsOnlyTarget only measures.

Run A span of text within a paragraph carrying its own style, link, footnote or language.

Section A run of pages with its own page setup; a document is one implicit section (blocks) or several explicit ones (sections).

Spot color A named ink (e.g. a Pantone) emitted as a PDF /Separation color space with a CMYK alternate; see Color Spaces.

Story A free-standing list of blocks, not in the normal flow, poured into a thread of frames.

Structure tree The logical tree of elements (Document, H1, P, Table, …) that makes a PDF tagged and accessible.

Tagged PDF A PDF carrying a logical structure tree (the basis of PDF/UA accessibility).

Index

A
artifact 13, 16

B
barcode 12, 24
bleed 5, 16

C
CMYK 5, 16, 21–22
command line 7
counting pass 1, 18

E
embed 12, 14, 18

F
frame 1, 5, 20, 23

I
ICC 5, 16, 22, 24

J
JSON Pointer 11–12

L
layout map 1, 8–9, 11

M
MCID 13–14

O
OutputIntent 16, 21
overprint 5, 17, 22

P
page number 1
ParentTree 13, 15
PDF/A 5, 16, 21, 23–24
PDF/UA 3, 5, 14–16, 18, 23
position 1, 12

S
schema 5–6, 8, 16, 24
server 6, 9–10, 12, 18–19, 24
spot 5, 17
story 1, 20
structure tree 13, 18, 23
stylesheet 5, 23
subset 4, 14

T
thread 23
two-pass 1–2, 9

V
veraPDF 14–15

X
XMP 5–6, 13, 16, 21–23