ksTFL

Key Statistics

Tables, Figures, Listings

The R-Package to Create Submission-Ready Clinical Reports

Igor Aleschenkov, Vladimir Larchenko (a ksTFL Team)

KeyStat Solutions | keystatsolutions.com

The Problem We All Face

When working in R as biostatisticians and statistical programmers we face a daily challenge:

We have: Clean, planar statistical output dataframe

We need: Professional tables with hierarchical layouts, grouped sections, deduplication, and consistent styling for regulatory submissions

Traditional approach: Either manually format in Word (hours of tedious work), or write complex data manipulation code that creates "pre-formatted" data frames with merged cells, spacing rows, pagination, and formatting markers (flextable and similar)

The cost: Data integrity risks, maintenance nightmares, and countless hours of manual work. Any data update means redoing all formatting.

The ksTFL Solution

Core principle: Complete separation of data and presentation.

Your statistical datasets remain **clean, planar, validation- and reporting-ready**. All layout formatting happens at render time through declarative data-related specifications.

Key benefits:

✓ Change presentation without touching data
✓ Reproduce outputs reliably from same source
✓ One clean dataset → multiple output formats
✓ Easy QC: compare planar data, not formatted/restructured data
✓ No data integrity risks from formatting

How it works:

You describe what you want the output to look like using a simple, chainable declarative language.

ksTFL applies transformations during rendering, leaving your source data completely untouched.

Think of it as CSS for statistical tables—separation of content and presentation.

Wait, Why Reinvent the Wheel?

One may reasonably say:

“Wait a second. Don’t we already have flextable?”

Yes. And that is a fair question.

- `flextable` is a strong, flexible, general-purpose table toolkit.
- If you need one polished publication table, it can be a perfectly good choice.
- ksTFL is not trying to replace every reporting package in R.

What ksTFL is built for is narrower and more painful: clinical TFL production, DOCX-first output, deterministic pagination, and report factories where the same structure must be generated again and again without turning clean data into display data.

> The real question is not “Can flextable make a nice table?” It can. The real question is: “What happens when we need to generate hundreds of submission-style outputs quickly, reproducibly, and safely?”

So let us be fair to both tools and start from the same place: the plain summary table coming out of the analysis.

What Both Tools Actually Receive

Before any formatting, both tools start from the same kind of input: a boring, trustworthy, planar summary table.

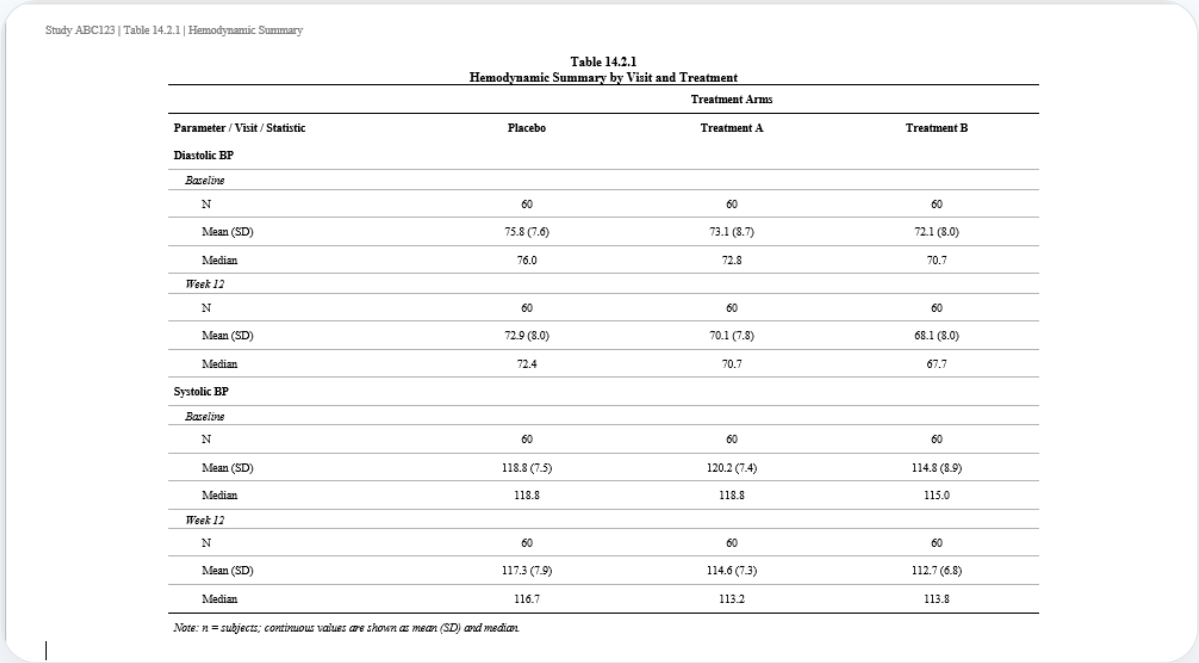
That matters, because once the input is the same, the comparison is no longer about statistics. It becomes a question of what each package asks you to do next.

PARAM	VISIT	STAT	Placebo	Treatment A	Treatment B
Diastolic BP	Baseline	N	60	60	60
Diastolic BP	Baseline	Mean (SD)	75.8 (7.6)	73.1 (8.7)	72.1 (8.0)
Diastolic BP	Baseline	Median	76.0	72.8	70.7
Diastolic BP	Week 12	N	60	60	60
Diastolic BP	Week 12	Mean (SD)	72.9 (8.0)	70.1 (7.8)	68.1 (8.0)
Diastolic BP	Week 12	Median	72.4	70.7	67.7
Systolic BP	Baseline	N	60	60	60
Systolic BP	Baseline	Mean (SD)	118.8 (7.5)	120.2 (7.4)	114.8 (8.9)

> This is what we want upstream: one row = one statistic, no fake spacer rows, no merged-cell placeholders, no formatting markers hidden in the data.

Same Destination, Different Trip

flextable



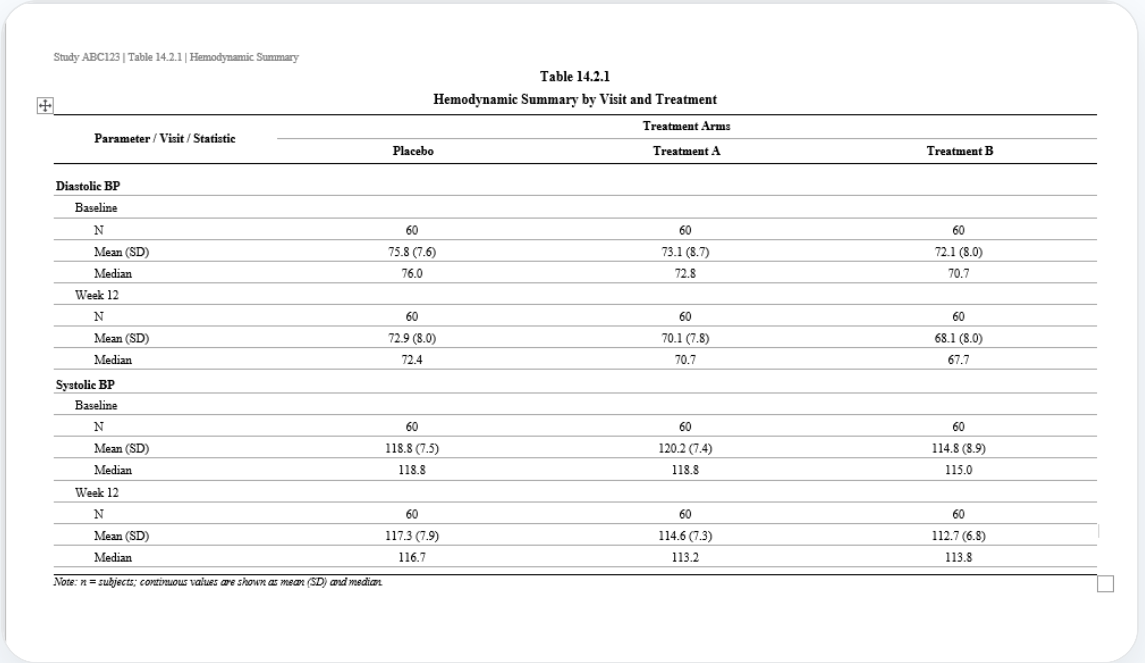
Study ABC123 | Table 14.2.1 | Hemodynamic Summary

Table 14.2.1
Hemodynamic Summary by Visit and Treatment

	Treatment Arms		
Parameter / Visit / Statistic	**Placebo**	**Treatment A**	**Treatment B**
Diastolic BP			
Baseline			
N	60	60	60
Mean (SD)	75.8 (7.6)	73.1 (8.7)	72.1 (8.0)
Median	76.0	72.8	70.7
Week 12			
N	60	60	60
Mean (SD)	72.9 (8.0)	70.1 (7.8)	68.1 (8.0)
Median	72.4	70.7	67.7
Systolic BP			
Baseline			
N	60	60	60
Mean (SD)	118.8 (7.5)	120.2 (7.4)	114.8 (8.9)
Median	118.8	118.8	115.0
Week 12			
N	60	60	60
Mean (SD)	117.3 (7.9)	114.6 (7.3)	112.7 (6.8)
Median	116.7	113.2	113.8

Note: n = subjects; continuous values are shown as mean (SD) and median.

ksTFL



Study ABC123 | Table 14.2.1 | Hemodynamic Summary

Table 14.2.1
Hemodynamic Summary by Visit and Treatment

	Treatment Arms		
Parameter / Visit / Statistic	**Placebo**	**Treatment A**	**Treatment B**
Diastolic BP			
Baseline			
N	60	60	60
Mean (SD)	75.8 (7.6)	73.1 (8.7)	72.1 (8.0)
Median	76.0	72.8	70.7
Week 12			
N	60	60	60
Mean (SD)	72.9 (8.0)	70.1 (7.8)	68.1 (8.0)
Median	72.4	70.7	67.7
Systolic BP			
Baseline			
N	60	60	60
Mean (SD)	118.8 (7.5)	120.2 (7.4)	114.8 (8.9)
Median	118.8	118.8	115.0
Week 12			
N	60	60	60
Mean (SD)	117.3 (7.9)	114.6 (7.3)	112.7 (6.8)
Median	116.7	113.2	113.8

Note: n = subjects; continuous values are shown as mean (SD) and median.

Yes, both can be made to produce a very similar DOCX table. That is not the interesting part. **The interesting part is the path.**

What the Two Paths Look Like

flextable best-case path

```
ft_grouped <- summary_tbl |>
  as_grouped_data(groups = c("PARAM", "VISIT"), ...)

ft_grouped <- ft_grouped[keep_rows, , drop = FALSE]

ft <- as_flextable(ft_grouped, ...)
ft <- set_header_labels(ft, ...)
ft <- add_header_row(ft, ...)
ft <- bold(ft, i = ~ ...PARAM rows..., ...)
ft <- italic(ft, i = ~ ...VISIT rows..., ...)
ft <- padding(ft, i = ~ ...STAT rows..., ...)
...
doc_ft <- read_docx()
doc_ft <- body_add_flextable(doc_ft, value = ft)
doc_ft <- body_set_default_section(doc_ft, ...)
```

- Restructure the display data.
- Clean up helper output.
- Style the table row by row.
- Assemble the Word document separately through `officer`.

ksTFL path

```
spec <- create_table(summary_tbl) |>
  define_cols(c(PARAM, VISIT), isVisible = FALSE) |>
  define_cols(c(STAT, Placebo, Treatment_A, Treatment_B), ...) |>
  add_span_header(c(Placebo, Treatment_A, Treatment_B), ...) |>
  add_title(...) |>
  add_header(...) |>
  add_footer(...) |>
  add_footnote(...) |>
  compute_cols(firstOf(PARAM), c_addrow("above", value_from = PARAM, ...)) |>
  compute_cols(firstOf(PARAM, VISIT), c_addrow("above", value_from = VISIT, ...))

write_doc(create_report(spec), ...)
```

- Keep `summary_tbl` exactly as it is.
- Describe the hierarchy declaratively.
- Render the DOCX through one reporting pipeline.

Let's Zoom In on the flextable Path

To be fair, this is the best native flextable route we found. And the very first step is not styling. It is reshaping the clean summary into a display-oriented object.

```
ft_grouped <- summary_tbl |>
  mutate(STAT = as.character(STAT)) |>
  as_grouped_data(
    groups = c("PARAM", "VISIT"),
    columns = c("STAT", "Placebo", "Treatment_A", "Treatment_B")
  )

# extra cleanup: the helper repeats PARAM rows
param_only_rows <- !is.na(ft_grouped$PARAM) &
  is.na(ft_grouped$VISIT) & is.na(ft_grouped$STAT)
keep_rows <- !(param_only_rows & duplicated(ft_grouped$PARAM))
ft_grouped <- ft_grouped[keep_rows, , drop = FALSE]
```

- We start from clean `summary_tbl`.
- Then we convert it into a grouped display object.
- Then we clean up the grouped display object because it still does not match the target hierarchy exactly.

At this point we have not improved the statistics. We have only taught the output how to look hierarchical.

Then Comes the Styling Surgery

Now that the display object exists, we still need to tell flextable which rows are PARAM rows, which are VISIT rows, which are detail rows, how much to indent them, how to align them, and how to draw the borders.

```
ft <- as_flextable(
  ft_grouped,
  hide_grouplabel = TRUE,
  col_keys = c("STAT", "Placebo", "Treatment_A", "Treatment_B")
)

ft <- set_header_labels(ft, STAT = "Parameter / Visit / Statistic",
  Placebo = "Placebo", Treatment_A = "Treatment A", Treatment_B = "Treatment B")
ft <- add_header_row(ft, values = c("", "Treatment Arms"), colwidths = c(1, 3))
ft <- add_footer_lines(ft, values = table_note)
ft <- border_remove(ft)
ft <- font(ft, fontname = "Times New Roman", part = "all")
ft <- fontsize(ft, size = 8, part = "all")
ft <- bold(ft, i = ~ !is.na(PARAM) & is.na(VISIT) & is.na(Placebo) &
  is.na(Treatment_A) & is.na(Treatment_B), j = "STAT", part = "body")
ft <- italic(ft, i = ~ !is.na(VISIT) & is.na(Placebo) &
  is.na(Treatment_A) & is.na(Treatment_B), j = "STAT", part = "body")
ft <- padding(ft, i = ~ !is.na(VISIT) & is.na(Placebo) &
  is.na(Treatment_A) & is.na(Treatment_B), j = "STAT", padding.left = 12, ...)
ft <- padding(ft, i = ~ !is.na(STAT), j = "STAT", padding.left = 24, ...)
ft <- align(ft, j = "STAT", align = "left", part = "all")
ft <- align(ft, j = c("Placebo", "Treatment_A", "Treatment_B"), align = "center", part = "all")
```

- This is where the code becomes output surgery.
- The table is styled by row type formulas, not by report semantics.
- **We are still not done. We only have the table object.**

And Then, Finally, Word Assembly

One more twist: flextable is not the whole DOCX story. To get headers, footers, landscape layout, and final export, we need to switch mental models and assemble the Word document separately through `officer`.

```
header_block <- block_list(fpar(
  ftext(doc_header_text, fp_text_lite(font.size = 8, font.family = "Times New Roman")),
  fp_p = fp_par(text.align = "left")))

footer_block <- block_list(fpar(
  ftext(doc_footer_text, fp_text_lite(font.size = 8, font.family = "Times New Roman")),
  fp_p = fp_par(text.align = "left")))

landscape_section <- prop_section(
  page_size = page_size(orient = "landscape"),
  page_margins = page_mar(...),
  type = "continuous",
  header_default = header_block,
  footer_default = footer_block
)

doc_ft <- read_docx()
doc_ft <- body_add_fpar(doc_ft, value = title_line_1)
doc_ft <- body_add_fpar(doc_ft, value = title_line_2)
doc_ft <- body_add_flextable(doc_ft, value = ft)
doc_ft <- body_set_default_section(doc_ft, value = landscape_section)
print(doc_ft, target = file.path(out_dir, "complex_docx_flextable.docx"))
```

- So the full solution is not “just flextable”. It is `flextable` plus a separate `officer` assembly layer.
- This is why the pain is real: more objects, more steps, more room for drift.
- **Friendly conclusion: yes, it works. But it asks a lot from the user.**

Now Compare That With the Full ksTFL Path

Now let us take exactly the same `summary_tbl` and ask ksTFL to produce the same submission-style table, but through one reporting language instead of a chain of output repairs.

```
library(ksTFL)

tfl_set_options(docTemplate = "Classic_landscape_times")

spec <- create_table(summary_tbl) |>
  define_cols(c(PARAM, VISIT), isVisible = FALSE) |>
  define_cols(
    c(STAT, Placebo, Treatment_A, Treatment_B),
    label = c( "Parameter / Visit / Statistic", "Placebo", "Treatment A", "Treatment B"),
    valueStyleRef = c("indent_2", "ac", "ac", "ac")
  ) |>
  add_span_header(
    c(Placebo, Treatment_A, Treatment_B),
    label = "Treatment Arms"
  ) |>
  add_title(table_title_1) |>
  add_title(table_title_2) |>
  add_header(doc_header_text) |>
  add_footer(doc_footer_text) |>
  add_footnote(table_note)
```

Up to this point, the code is still talking only about the report: columns, labels, titles, header/footer, and footnote. Nothing has been restructured yet.

ksTFL Full Path, Continued

Now let's modify layout

```
spec <- spec |>
  compute_cols(
    firstOf(PARAM),
    c_addrow("above", value_from = PARAM, styleRef = "grp_hdr")
  ) |>
  compute_cols(
    firstOf(PARAM, VISIT),
    c_addrow("above", value_from = VISIT, styleRef = "indent_1")
  )

report_obj <- create_report(spec)

write_doc(
  report_obj,
  name = "complex_docx_kstfl",
  outDir = out_dir,
  metaPath = tempdir()
)
```

- This is the whole path.
- The hierarchy levels are declared from data boundaries with conditional logic.
- Rendering, layout, and pagination all stay inside one reporting pipeline.
- That is the real contrast: ksTFL spends its code on report logic, not on output choreography.

What ksTFL Is Really Buying You

Simpler reporting logic

- Hidden grouping columns stay as data, not display rows.
- Hierarchy comes from conditions, like `firstOf(...)`, and actions, like `c_addrow(...)` - not manual reshaping.
- Headers, footnotes, layout, and rendering stay in one DSL.

Less output surgery

- No separate grouped display object.
- No cleanup of duplicated hierarchy rows.
- No second mental model for `officer` document assembly.
- Code reads more like “describe the report” and less like “repair the table.”

That is why ksTFL feels “smaller” in code for this job. It removes output choreography that general-purpose table tools leave to the user.

And Then There Is Performance

So far, we have talked about code pain. But maybe one could still say: **“Fine, I can survive ugly code if the result is worth it.”**

So let us take the same kind of planar summary as above, keep the table structure the same, and simply make it much larger. This is where convenience stops being the main question and performance starts becoming a production issue.

Input shape	Summary rows	Result	flextable	ksTFL
50 PARAM x 20 VISIT	3000	32-page DOCX	27.95 sec	1.37 sec
50 PARAM x 100 VISIT	15000	large DOCX completed	459.86 sec	5.31 sec
50 PARAM x 200 VISIT	30000	flextable failed	**> 30 min + memory error**	8.59 sec

Human translation: 27 seconds is already coffee-break territory. 459 seconds is “oh no...” territory. And “more than 30 minutes, then memory error” is the point where nobody calmly says, “yes, this pipeline scales nicely.”

> This is a synthetic stress test to show scaling behavior. Real studies will not have this exact shape, but they absolutely do produce tables and listings large enough for runtime and memory behavior to become a real workflow problem.

Fast Is Nice. Predictable Is Better.

And this is why the previous slide matters so much: those ksTFL timings — 1.37 seconds, 5.31 seconds, even 8.59 seconds for the largest successful case — are not just “file saved” timings.

Within those few seconds, ksTFL is not merely dumping a table into a DOCX file.

It is doing real layout work that many users assume Word will somehow figure out later.

It is also:

- **paginating the output itself instead of asking Word to figure it out later**
- **knowing in advance how many pages the document will have**
- **detecting when the selected layout cannot fit part of the table correctly**

In clinical reporting, that matters.
Faster output is useful. Deterministic, validated, page-aware output is what saves time in production.

And What About Reporter?

There are many reporting packages in R, and they all solve the problem in their own way. The nearest declarative, relatively simple package to ksTFL is probably **SASSY Reporter**, and it is definitely a nice package.

But when creating ksTFL, we were trying to avoid several limitations we experienced with Reporter:

- Reporter was originally built with English-first workflows in mind; in our experience, full UTF and multilingual output, especially when correct pagination matters, can become painful.
- Support for superscripts, subscripts, and richer text formatting is more limited than what we wanted.
- Styling possibilities are more limited than what we wanted for submission-quality DOCX output.
- Some layouts, especially indented categories and stubs, can require the data to be prepared in a Reporter-specific structure first.
- Reporter performs better than flextable, but on very large outputs it was still not where we wanted it to be.
- And when those outputs are large RTF files, MS Word itself can become part of the pain as well.

This is not criticism of the Reporter package. On the contrary, it is still one of the best R packages for producing clinical TFLs, and it is much closer to ksTFL in spirit than general-purpose table toolkits. ksTFL was created because we wanted to go further on multilingual DOCX rendering, pagination control, styling depth, and very large-output performance.

Now that we know why ksTFL exists, let's go back to a small example and learn the language step by step.

Our Source Data

Here is typical output from a demographics analysis. Notice: it is **clean, tidy, and planar**—exactly what good statistical programming usually produces.

PARAM	STATISTICS	TRTA	TRTB
Age (years)	n	10	15
Age (years)	Mean (SD)	46.9 (14.6)	48.2 (15.2)
Age (years)	Median	47	48
Sex	Female	5 (50.0%)	7 (46.7%)
Sex	Male	5 (50.0%)	8 (53.3%)

Why this structure matters: Each row is one statistic. Columns are variables, not formatted headers. This structure works with standard dplyr/tidyverse tools, can be validated and tested easily, is version-controllable and diff-friendly, and doesn't mix data with presentation.

First Render: What Happens with Zero Configuration?

Let's start with the absolute minimum code and see what ksTFL gives us out of the box:

Step 1: Load the package

```
library(ksTFL)
```

> Import ksTFL functions into your R session

Step 2: Create table specification

```
spec <- create_table(demog_1_1)
```

> This copies your data into spec's internal environment (immutable snapshot), auto-detects column types, calculates default widths, and creates the rendering blueprint

Step 3: Create report object

```
report <- create_report(spec)
```

> Assembles one or more specs into a report. You can combine multiple tables, figures, and text blocks here

Step 4: Write to document

```
write_doc(report, '1_1_0_table')
```

> Renders the report to DOCX format. The C++ engine processes all transformations and generates the Word document

PARAM	STATISTICS	TRTA	TRTB
Age (years)	n	10	15
Age (years)	Mean (SD)	46.9 (14.6)	48.2 (15.2)
Age (years)	Median	47	48
Sex	Female	5 (50.0%)	7 (46.7%)
Sex	Male	5 (50.0%)	8 (53.3%)

What we got: a perfectly honest plain data listing. All the data is there, but it is not submission-ready yet. So let's add professional polish, step by step.

📖 **create_table() | create_report() | write_doc()**

Step 1: Making Headers Readable

Column names like `PARAM` and `TRTA` are great for programming, but regulatory reviewers need descriptive labels. Let's make the headers human-friendly:

```
spec <- create_table(demog_1_1) |>
  define_cols(
    c(PARAM, STATISTICS, TRTA, TRTB),
    label = c('Parameter', 'Statistics',
              'Treatment A', 'Treatment B'),
    colWidth = c('3cm', '30%', '20%', '20%')
  )
```

What `define_cols()` does: Controls how columns appear in output—their labels, widths, visibility, alignment, and styling.

Why control widths? Mixed units let you fix critical columns (3cm for Parameter) while letting others flex proportionally (20% each for treatments).

You can call `define_cols()` once for all columns, or multiple times for fine-grained control.

📖 **define_cols() reference**

Parameter	Statistics	Treatment A	Treatment B
Age (years)	n	10	15
Age (years)	Mean (SD)	46.9 (14.6)	48.2 (15.2)
Age (years)	Median	47	48
Sex	Female	5 (50.0%)	7 (46.7%)
Sex	Male	5 (50.0%)	8 (53.3%)

> **Improvement**: Headers now make sense to reviewers. Column widths are controlled.
>
> **Remaining issue**: Repeating "Age (years)" and "Sex" values create visual clutter.

Step 2: Removing Visual Clutter

See how "Age (years)" repeats three times? This is not how we usually want a production table to look.
Let's clean that up with one simple flag:

```
spec <- spec |>
  define_cols( PARAM,  dedupe = TRUE  )
```

The `dedupe` feature: When consecutive cells in a column contain identical values, all but the first are visually cleared at render time.

Critical point: Your source data remains completely unchanged. This is a render-time transformation. You can turn it on or off, change it for different outputs, without ever reprocessing your statistical analysis.

Why this matters: One clean dataset can produce multiple presentations—detailed for appendices, deduplicated for summaries.

📖 **Column properties docs**

Parameter	Statistics	Treatment A	Treatment B
Age (years)	n	10	15
	Mean (SD)	46.9 (14.6)	48.2 (15.2)
	Median	47	48
Sex	Female	5 (50.0%)	7 (46.7%)
	Male	5 (50.0%)	8 (53.3%)

> **Improvement**: Parameter groups are now visually distinct. Much easier to read.
>
> **Next challenge**: Where does one parameter group end and the next begin? Professional tables use whitespace to guide the eye.

Step 3: Using Whitespace to Guide the Eye

Now the table is cleaner, but where does Age end and Sex begin? Professional tables use whitespace strategically to create visual boundaries. Let's add spacing between parameter groups:

```
spec <- spec |>
  compute_cols(
    firstOf(PARAM),
    c_addrow('above')
  )
```

Parameter	Statistics	Treatment A	Treatment B
Age (years)	n	10	15
	Mean (SD)	46.9 (14.6)	48.2 (15.2)
	Median	47	48
Sex	Female	5 (50.0%)	7 (46.7%)
	Male	5 (50.0%)	8 (53.3%)

Understanding `compute_cols()` : This is where ksTFL's power emerges. You specify two things:

1. A condition — `firstOf(PARAM)` identifies the first row of each parameter group. *`firstOf()` here is just one of the built-in helper functions. But you are free to write almost any R logical condition here, for example* *`PARAM == 'Sex' & STATISTICS == 'Female'`*.

2. An action — `c_addrow('above')` inserts an empty row before those matched rows

When it happens: During rendering, not during data processing. Your source data never changes.

Why this matters: You're declaratively describing visual structure, not imperatively manipulating data.

> **Improvement**: Clear visual breaks between sections. Parameter groups are obvious at a glance.
>
> **Can we go further?** Yes—submission-ready tables often use hierarchical layouts with parameters as section headers and statistics indented below.

📖 **compute_cols() | Condition helpers | Actions**

Step 4: Creating Submission-Standard Hierarchy

Ready for the final transformation? Submission-ready tables often use hierarchical layouts: parameters as bold section headers with statistics indented below. Here's the clever trick: **hide the PARAM column but keep it for logic**, then use its values to create those section headers:

```
spec <- create_table(demog_1_1) |>
  define_cols(PARAM, isVisible = FALSE) |>
  define_cols(
    c(STATISTICS, TRTA, TRTB),
    label = c('Parameter<br>  Statistics', 'Treatment A', 'Treatment B'),
    labelStyleRef = c('al', NA, NA),
    valueStyleRef = 'indent_1',
    colWidth = c(NA, '4cm', '4cm')
  ) |>
  compute_cols(
    firstOf(PARAM),
    c_addrow('above', value_from = PARAM, styleRef = 'b')
  ) |>
  set_document(contentWidth = '40%')
```

Breaking it down: PARAM is hidden from output with `isVisible = FALSE`, but remains available for conditional logic. We detect parameter boundaries with `firstOf(PARAM)` helper function, then insert section header rows using `value_from = PARAM`. Statistics and values get 5mm indent with `'indent_1'`, headers get bold with `'b'` embedded styles.

📖 **set_document() | Action parameters**

The Result: Submission-Ready Layout

Parameter Statistics	Treatment A	Treatment B
Age (years)		
n	10	15
Mean (SD)	46.9 (14.6)	48.2 (15.2)
Median	47	48
Sex		
Female	5 (50.0%)	7 (46.7%)
Male	5 (50.0%)	8 (53.3%)

What happened:

✓ PARAM values became bold section headers
✓ Statistics indented below each parameter
✓ Clean hierarchical structure
✓ Compact 40% page width

Atomic styles used:

- `'al'` — align left
- `'b'` — bold text
- `'indent_1'` — 5mm indent

The power: Same source data, completely different presentation. Change the spec, not the data.

View all built-in styles:
`tfl_print_style_atoms()`

Understanding the Magic Behind It

Let's pause and understand what just happened. This is where ksTFL's philosophy really shines:

Hiding without losing: `isVisible = FALSE` hides PARAM from output but keeps it available for logic. We use this hidden column to:

- Detect parameter boundaries with `firstOf(PARAM)`
- Insert section header rows with `value_from = PARAM`
- Apply bold styling to those headers with `styleRef = 'b'`

Why this matters: The same source data can drive multiple presentation formats. Need a different layout? Change the spec, not the data. Need both flat and hierarchical views? Create two specs from one dataset.

Atomic styles: ksTFL includes built-in styles for common needs. No need to define styles for basic formatting (but you definitely can if you want to):

- Alignment: `'al'`, `'ac'`, `'ar'`
- Text: `'b'`, `'i'`, `'u'`
- Indents: `'indent_1'` (5mm), `'indent_2'` (10mm)

See all atomic styles: `tfl_print_style_atoms()`

📖 **Built-in styles reference**

The Core Concepts

Three main functions control table rendering:

`define_cols()` — Column presentation properties:

- `label`: Header text (supports HTML: `<br>`, `<b>`, `<i>`)
- `colWidth`: Fixed (`'3cm'`) or percentage (`'20%'`)
- `isVisible`: Hide column but keep for logic
- `dedupe`: Remove consecutive duplicates visually
- `labelStyleRef`, `valueStyleRef`: Apply formatting

`compute_cols()` — Conditional transformations:

- **Condition helpers**: `firstOf()`, `lastOf()`, `rowNum()`, ..., **or almost any R expression**
- **Actions to perform on matching rows**: `c_addrow()`, `c_style()`, `c_merge()`, `c_pageBreak()`
- Actions fire when conditions match, during report assembly

`set_document()` — Overall layout control:

- `contentWidth`: Table width as percentage or dimension
- Document-level settings: orientation, margins, etc.

📖 **define_cols() | compute_cols() | set_document()**

The Final Touch: Adding Document Metadata

Here's the thing about regulatory submissions: A table without proper context gets rejected. You need titles, footnotes, headers, footers—the full documentation trail.

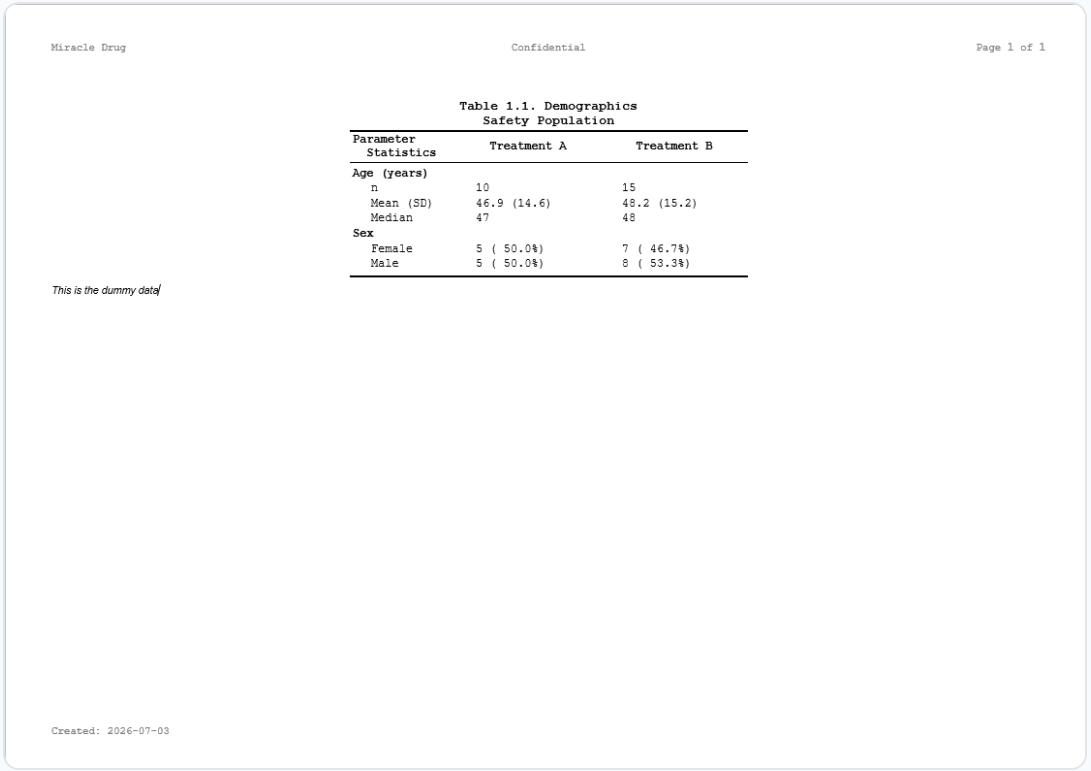
Let's add that final layer of metadata that takes our beautifully formatted table and makes it submission-ready:

```
spec <- spec |>
  add_title(c("Table 1.1. Demographics",
              "Safety Population")) |>
  add_footnote("This is the dummy data") |>
  add_header(c("Miracle Drug",
               "Confidential",
               "Page {PAGE} of {NUMPAGES}")) |>
  add_footer(paste0("Created: ", Sys.Date()))

report <- create_report(spec)
write_doc(report, '1_1_5_table')
```

What each function does: `add_title()` provides table identification and population. `add_footnote()` adds data context and clarifications. `add_header()` creates running headers with dynamic placeholders. `add_footer()` adds traceability information like creation dates.

And We're Done! A Complete Submission Document



Miracle Drug Confidential Page 1 of 1

Table 1.1. Demographics
Safety Population

Parameter Statistics	Treatment A	Treatment B
Age (years)		
n	10	15
Mean (SD)	46.9 (14.6)	48.2 (15.2)
Median	47	48
Sex		
Female	5 (50.0%)	7 (46.7%)
Male	5 (50.0%)	8 (53.3%)

This is the dummy data

Created: 2026-07-03

From plain dataframe to this in just a few lines of code:

✓ **Running headers**: Study name, confidentiality, page numbers
✓ **Table title**: Proper identification with population specification
✓ **Formatted content**: Hierarchical layout with proper styling
✓ **Footnotes**: Context and data source information
✓ **Footer**: Creation timestamp for traceability

Dynamic placeholders: Use `{PAGE}`, `{NUMPAGES}`. They're replaced at render time with actual values.

Multi-line support: All metadata functions accept character vectors—each element becomes a separate line.

📖 **Titles** | **Footnotes** | **Headers** | **Footers**

The Philosophy: Declarative Transformation

Understanding the paradigm shift:

Traditional approach: "Prepare my data to look like the output"

- Mix data with formatting artifacts (spacing rows, indents in texts)
- Lose ability to easily validate and compare datasets
- Hard to maintain—changes require data reprocessing
- Difficult to review—what changed, data or format?

ksTFL approach: "Describe what I want the output to look like"

- Data stays clean and validation-ready
- Presentation is a separate, versioned layer

Think in transformations: You're not formatting data, you're declaring rendering rules that the system applies consistently every time.

This separation enables reproducibility, version control, and confidence in regulatory submissions.

What This Enables in Practice

For routine work:
Define table structures once as templates, reuse across studies. Consistent formatting applied automatically to new data.

For complex tables:
Build progressively, testing each refinement. No risk of corrupting source data during experimentation.

For submission packages:
Reproduce exact outputs from validated data. Your audit trail shows what changed—spec version, not data manipulation.

For collaboration:
Statisticians create clean output. Stat programmers refine presentation. Both work with the same data, different layers.

For QC:
Compare planar, not formatted data frames. Much easier to spot intentional design changes vs. accidental data errors.

For sponsors:
You can easily change the document template, control the table layout (paginated vs. continuous), control footnote placement, and much more without touching your input data, and sometimes even without rebuilding the data.

Beyond the Basics

The simple demographics example showed core concepts. But ksTFL handles much more complex scenarios:

Spanning headers: Multi-level column groupings with `add_span_header()` for complex multi-arm trials

Grouping: Split single dataframes into multiple pages with `isGrouping`

Dynamic content: Use `#ByGroupX` placeholders in subtitles

Text manipulation: Combine multiple values in cells with `c_glue()`

Smart selectors: Use tidyselect helpers for efficient column operations

Table of contents: Multi-table documents with automatic navigation, and much more

📖 **Spanning headers** | **Page breaks** | **Column breaks** | **Style actions** | **Style system**

Let's see these features in action with a real clinical trial example...

Real Challenge: Multi-Population Lab Changes Table

The scenario: You're tracking lab parameters across visits. You need to report both change-from-screening AND change-from-baseline. And you need separate tables for Safety and Per-Protocol populations.
Let's imagine we calculated all the statistics and prepared a single dataframe like this:

The data structure: 15 columns in our dataframe —

- `POPULATION` : Safety vs. Per-Protocol (grouping variable)
- `PARAMETER` , `VISIT` : Row identifiers
- **Observed values**: `TRT_A_OBS_MSD` , `TRT_A_OBS_MEDIAN` , `TRT_B_OBS_MSD` , `TRT_B_OBS_MEDIAN`
- **Changes from Screening**: `TRT_A_CHG_S_MSD` , `TRT_A_CHG_S_MEDIAN` , `TRT_B_CHG_S_MSD` , `TRT_B_CHG_S_MEDIAN`
- **Changes from Baseline**: `TRT_A_CHG_B_MSD` , `TRT_A_CHG_B_MEDIAN` , `TRT_B_CHG_B_MSD` , `TRT_B_CHG_B_MEDIAN`

POPULATION	PARAMETER	VISIT	TRT_A_OBS_MSD	TRT_A_OBS_MEDIAN	TRT_B_OBS_MSD	TRT_B_OBS_MEDIAN	TRT_A_CHG_S_MSD	TRT_A_CHG_S_MEDIAN	TRT_B_CHG_S_MSD	TRT_B_CHG_S_MEDIAN	TRT_A_CHG_
Safety	ALT (U/L)	Screening	41.8 (12.3)	40	43.5 (13.1)	42					
Safety	ALT (U/L)	Baseline	40.6 (11.8)	39	42.8 (12.7)	41.5					
Safety	ALT (U/L)	Visit 1	36.8 (10.9)	35.8	40.4 (12.0)	39.6	-5.0 (6.8)	-4.6	-3.1 (7.2)	-2.8	-3.8 (5.6)
Per-Protocol	Hemoglobin (g/L)	Visit 2	140.4 (9.7)	140.5	138.2 (9.8)	138	5.4 (5.3)	5.2	4.2 (5.4)	4	4.6 (4.7)

The challenge: Notice how Screening/Baseline rows have empty CHG columns (no changes yet), while Visit rows populate all 15 columns. We need to transform this wide, multi-population dataframe into professional multi-page output.

The goal: Two separate table pages (one per population) with three-level spanning headers, parameter section breaks, and dynamic subtitles.

Smart Move: Set Global Defaults First

Pro tip: When generating multiple tables for a submission package, don't repeat yourself. Set common settings once at the session level—headers, footers, templates—and they apply to every table you create:

```
# Common document headers and footers
tfl_set_options(
  add_header(c("Miracle Drug", "Confidential", "Page {PAGE} of {NUMPAGES}")),
  add_footer(paste0("Created: ", Sys.Date()))
)

# Switch to landscape template with Times New Roman font
# Just to show how easy it is to switch to another table view
tfl_set_options(docTemplate = 'Classic_landscape_times')
```

Why use package options? When generating multiple tables for a submission, you want consistent branding and layout. Set once, apply everywhere.

Built-in templates: ksTFL includes several pre-configured templates. You can also create custom templates using the package's Shiny add-in for complete control.

📖 **tfl_set_options() | tfl_list_templates()**

Step 1: Set Up Grouping (Remember This From Before?)

The familiar pattern: Just like we hid PARAM in the demographics table, we'll use the same strategy here. But this time, we add something new—`isGrouping`:

```
lab_spec <- create_table(lbchg_2_1) |>
  # POPULATION splits into separate pages but doesn't appear in the table
  define_cols(POPULATION, isVisible = FALSE, isGrouping = TRUE) |>
  # PARAMETER will become section headers (same trick as before)
  # VISIT stays visible as row identifier
  define_cols(
    c(PARAMETER, VISIT),
    label = c(NA, 'Visit'),
    isVisible = c(FALSE, NA)
  )
```

New concept—`isGrouping`: This creates separate pages for each unique POPULATION value. Same table structure, different data subset. One page for Safety, one for Per-Protocol. Automatic.

📖 **Grouping columns docs**

Step 2: Work Smarter with Pattern Matching

Imagine labeling 12 columns one by one. Tedious, right? But look at your column names—they follow patterns! `TRT_A_OBS_MSD`, `TRT_B_CHG_S_MEDIAN`... Let's exploit that structure:

```
lab_spec <- lab_spec |>
  define_cols(contains('_MEDIAN'), label = 'Median') |>
  define_cols(contains('_MSD'), label = 'Mean (SD)')
```

Why this matters: If your statistical programming team uses consistent naming conventions (and they should!), you can label dozens of columns with just a few lines. Less code = fewer typos = easier maintenance.

The result: All 12 statistical columns get proper labels in two function calls instead of twelve.

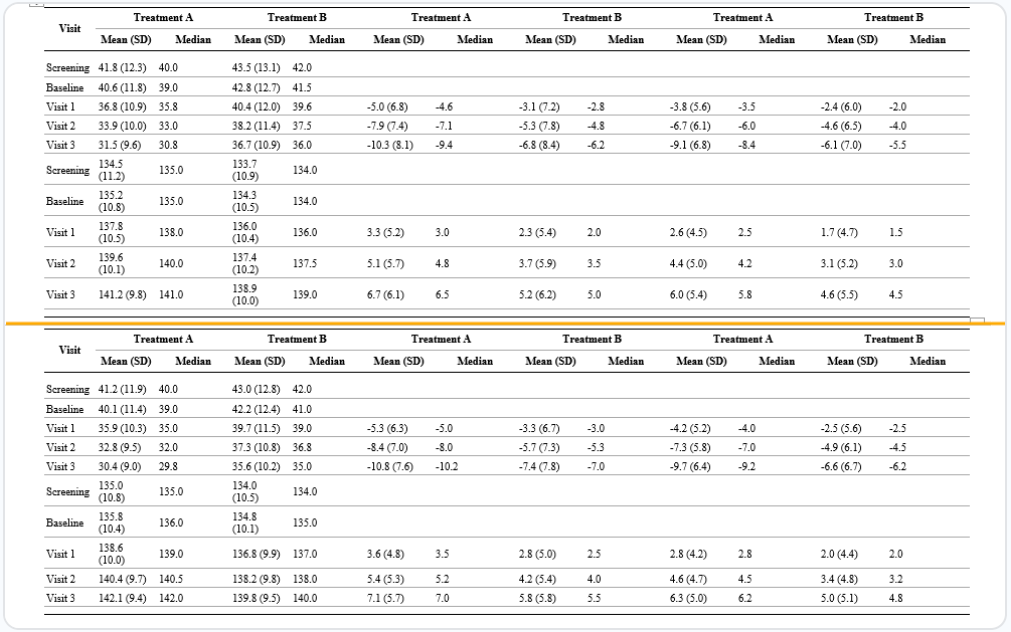
📖 **tidyselect helpers reference**

Step 3: Building the Header Hierarchy (Part 1)

Now for the spanning headers. We need three levels: treatment (A/B) → value type (Observed/Change) → statistic (Mean/Median). Let's build it layer by layer, starting with treatments:

```
lab_spec <- lab_spec |>
  add_span_header(
    starts_with('TRT_A_OBS'),
    'Treatment A'
  ) |>
  add_span_header(
    starts_with('TRT_B_OBS'),
    'Treatment B',
    stubOrder = 1
  ) |>
  add_span_header(
    starts_with('TRT_A_CHG_S'),
    'Treatment A',
    stubOrder = 1
  ) |>
  # ... and so on for CHG_B columns
```

What this does: Groups columns by treatment arm. Each treatment gets a spanning header covering its statistics columns.



Visit	Treatment A		Treatment B		Treatment A		Treatment B		Treatment A		Treatment B	
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
Screening	41.8 (12.3)	40.0	43.5 (13.1)	42.0								
Baseline	40.6 (11.8)	39.0	42.8 (12.7)	41.5								
Visit 1	36.8 (10.9)	35.8	40.4 (12.0)	39.6	-5.0 (6.8)	-4.6	-3.1 (7.2)	-2.8	-3.8 (5.6)	-3.5	-2.4 (6.0)	-2.0
Visit 2	33.9 (10.0)	33.0	38.2 (11.4)	37.5	-7.9 (7.4)	-7.1	-5.3 (7.8)	-4.8	-6.7 (6.1)	-6.0	-4.6 (6.5)	-4.0
Visit 3	31.5 (9.6)	30.8	36.7 (10.9)	36.0	-10.3 (8.1)	-9.4	-6.8 (8.4)	-6.2	-9.1 (6.8)	-8.4	-6.1 (7.0)	-5.5
Screening	134.5 (11.2)	135.0	133.7 (10.9)	134.0								
Baseline	135.2 (10.8)	135.0	134.3 (10.5)	134.0								
Visit 1	137.8 (10.5)	138.0	136.0 (10.4)	136.0	3.3 (5.2)	3.0	2.3 (5.4)	2.0	2.6 (4.5)	2.5	1.7 (4.7)	1.5
Visit 2	139.6 (10.1)	140.0	137.4 (10.2)	137.5	5.1 (5.7)	4.8	3.7 (5.9)	3.5	4.4 (5.0)	4.2	3.1 (5.2)	3.0
Visit 3	141.2 (9.8)	141.0	138.9 (10.0)	139.0	6.7 (6.1)	6.5	5.2 (6.2)	5.0	6.0 (5.4)	5.8	4.6 (5.5)	4.5

Visit	Treatment A		Treatment B		Treatment A		Treatment B		Treatment A		Treatment B	
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
Screening	41.2 (11.9)	40.0	43.0 (12.8)	42.0								
Baseline	40.1 (11.4)	39.0	42.2 (12.4)	41.0								
Visit 1	35.9 (10.3)	35.0	39.7 (11.5)	39.0	-5.3 (6.3)	-5.0	-3.3 (6.7)	-3.0	-4.2 (5.2)	-4.0	-2.5 (5.6)	-2.5
Visit 2	32.8 (9.5)	32.0	37.3 (10.8)	36.8	-8.4 (7.0)	-8.0	-5.7 (7.3)	-5.3	-7.3 (5.8)	-7.0	-4.9 (6.1)	-4.5
Visit 3	30.4 (9.0)	29.8	35.6 (10.2)	35.0	-10.8 (7.6)	-10.2	-7.4 (7.8)	-7.0	-9.7 (6.4)	-9.2	-6.6 (6.7)	-6.2
Screening	135.0 (10.8)	135.0	134.0 (10.5)	134.0								
Baseline	135.8 (10.4)	136.0	134.8 (10.1)	135.0								
Visit 1	138.6 (10.0)	139.0	136.8 (9.9)	137.0	3.6 (4.8)	3.5	2.8 (5.0)	2.5	2.8 (4.2)	2.8	2.0 (4.4)	2.0
Visit 2	140.4 (9.7)	140.5	138.2 (9.8)	138.0	5.4 (5.3)	5.2	4.2 (5.4)	4.0	4.6 (4.7)	4.5	3.4 (4.8)	3.2
Visit 3	142.1 (9.4)	142.0	139.8 (9.5)	140.0	7.1 (5.7)	7.0	5.8 (5.8)	5.5	6.3 (5.0)	6.2	5.0 (5.1)	4.8

Current state: Two pages (Safety and Per-Protocol), treatment headers are visible, but the top-level headers are still missing. At this stage we also still do not see the parameter labels or the population name inside the table body.

Step 3: Building the Header Hierarchy (Part 2)

Now add the top level—distinguishing Observed values from Changes. Notice the `stubOrder = 2`? Higher numbers = higher in the hierarchy. This creates the three-level structure we need:

```
lab_spec <- lab_spec |>
  add_span_header(
    4:7,
    'Observed Values'
  ) |>
  add_span_header(
    8:11,
    'Change from Screening',
    stubOrder = 2
  ) |>
  add_span_header(
    12:15,
    'Change from Baseline',
    stubOrder = 2
  )
```

The `stubOrder` parameter: Controls hierarchy. Higher numbers = higher level. Creates three-level structure: type → treatment → statistic.

Table 2.1 Changes from Screening and Baseline in Laboratory Parameters

Safety Population

Visit	Observed Values				Change from Screening				Change from Baseline			
	Treatment A		Treatment B		Treatment A		Treatment B		Treatment A		Treatment B	
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
Screening	41.8 (12.3)	40.0	43.5 (13.1)	42.0								
Baseline	40.6 (11.8)	39.0	42.8 (12.7)	41.5								
Visit 1	36.8 (10.9)	35.8	40.4 (12.0)	39.6	-5.0 (6.8)	-4.6	-3.1 (7.2)	-2.8	-3.8 (5.6)	-3.5	-2.4 (6.0)	-2.0
Visit 2	33.9 (10.0)	33.0	38.2 (11.4)	37.5	-7.9 (7.4)	-7.1	-5.3 (7.8)	-4.8	-6.7 (6.1)	-6.0	-4.6 (6.5)	-4.0
Visit 3	31.5 (9.6)	30.8	36.7 (10.9)	36.0	-10.3 (8.1)	-9.4	-6.8 (8.4)	-6.2	-9.1 (6.8)	-8.4	-6.1 (7.0)	-5.5
Screening	134.5 (11.2)	135.0	133.7 (10.9)	134.0								
Baseline	135.2 (10.8)	135.0	134.3 (10.5)	134.0								
Visit 1	137.8 (10.5)	138.0	136.0 (10.4)	136.0	3.3 (5.2)	3.0	2.3 (5.4)	2.0	2.6 (4.5)	2.5	1.7 (4.7)	1.5
Visit 2	139.6 (10.1)	140.0	137.4 (10.2)	137.5	5.1 (5.7)	4.8	3.7 (5.9)	3.5	4.4 (5.0)	4.2	3.1 (5.2)	3.0
Visit 3	141.2 (9.8)	141.0	138.9 (10.0)	139.0	6.7 (6.1)	6.5	5.2 (6.2)	5.0	6.0 (5.4)	5.8	4.6 (5.5)	4.5

Table 2.1 Changes from Screening and Baseline in Laboratory Parameters

Per-Protocol Population

Visit	Observed Values				Change from Screening				Change from Baseline			
	Treatment A		Treatment B		Treatment A		Treatment B		Treatment A		Treatment B	
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
Screening	41.2 (11.9)	40.0	43.0 (12.8)	42.0								
Baseline	40.1 (11.4)	39.0	42.2 (12.4)	41.0								
Visit 1	35.9 (10.3)	35.0	39.7 (11.5)	39.0	-5.3 (6.3)	-5.0	-3.3 (6.7)	-3.0	-4.2 (5.2)	-4.0	-2.5 (5.6)	-2.5
Visit 2	32.8 (9.5)	32.0	37.3 (10.8)	36.8	-8.4 (7.0)	-8.0	-5.7 (7.3)	-5.3	-7.3 (5.8)	-7.0	-4.9 (6.1)	-4.5
Visit 3	30.4 (9.0)	29.8	35.6 (10.2)	35.0	-10.8 (7.6)	-10.2	-7.4 (7.8)	-7.0	-9.7 (6.4)	-9.2	-6.6 (6.7)	-6.2
Screening	135.0 (10.8)	135.0	134.0 (10.5)	134.0								
Baseline	135.8 (10.4)	136.0	134.8 (10.1)	135.0								
Visit 1	138.6 (10.0)	139.0	136.8 (9.9)	137.0	3.6 (4.8)	3.5	2.8 (5.0)	2.5	2.8 (4.2)	2.8	2.0 (4.4)	2.0
Visit 2	140.4 (9.7)	140.5	138.2 (9.8)	138.0	5.4 (5.3)	5.2	4.2 (5.4)	4.0	4.6 (4.7)	4.5	3.4 (4.8)	3.2
Visit 3	142.1 (9.4)	142.0	139.8 (9.5)	140.0	7.1 (5.7)	7.0	5.8 (5.8)	5.5	6.3 (5.0)	6.2	5.0 (5.1)	4.8

Progress check: Headers look great! But wait—where are the parameter names? We hid that column, remember? Let's bring those values back as bold section headers, just like we did in the demographics table.

Step 4: Add Metadata and Dynamic Content

Last step before rendering: Add titles, footnotes, and—here's something cool—dynamic subtitles that change per page:

```
lab_spec <- lab_spec |>
  add_title('Table 2.1 Changes from Screening and Baseline in Laboratory Parameters') |>
  add_subtitle('#ByGroup1 Population') |>  # Dynamic subtitle per group
  compute_cols(
    firstOf(PARAMETER),
    c_addrow('above', value_from = PARAMETER, styleRef = f_combine('b', 'ac'))
  )

report <- create_report(lab_spec)
write_doc(report, '2_1_1_table')
```

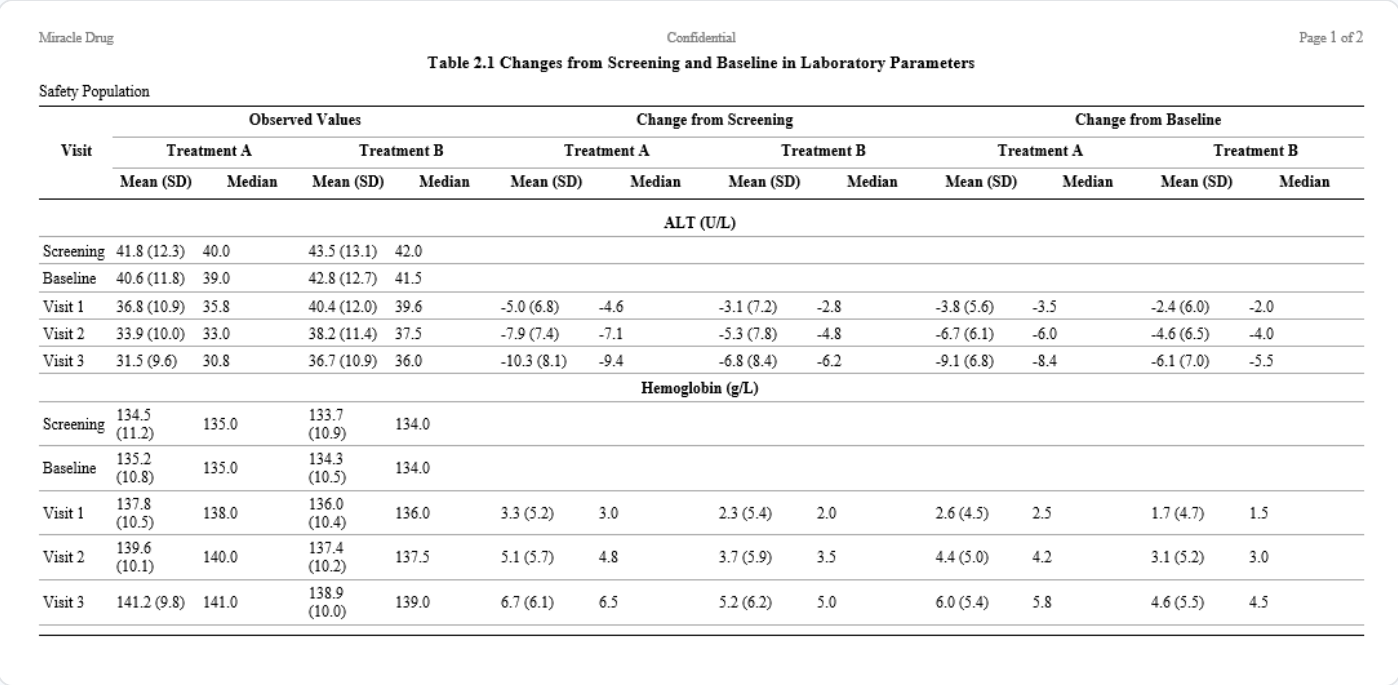
See that `#ByGroup1`? It's a placeholder that gets replaced at render time. Page 1 will say "Safety Population", page 2 will say "Per-Protocol Population"—automatically, from the same spec.

The `f_combine()` trick: Quickly merge atomic styles without defining new ones. Here: `'b'` (bold) + `'ac'` (align-center) = centered bold headers.

📖 **Dynamic content | f_combine()**

Look What We Built!

Here's the first page (Safety Population). The Per-Protocol page looks identical in structure, just different data:



Miracle Drug Confidential Page 1 of 2

Table 2.1 Changes from Screening and Baseline in Laboratory Parameters

Safety Population

Visit	Observed Values				Change from Screening				Change from Baseline			
	Treatment A		Treatment B		Treatment A		Treatment B		Treatment A		Treatment B	
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
ALT (U/L)												
Screening	41.8 (12.3)	40.0	43.5 (13.1)	42.0								
Baseline	40.6 (11.8)	39.0	42.8 (12.7)	41.5								
Visit 1	36.8 (10.9)	35.8	40.4 (12.0)	39.6	-5.0 (6.8)	-4.6	-3.1 (7.2)	-2.8	-3.8 (5.6)	-3.5	-2.4 (6.0)	-2.0
Visit 2	33.9 (10.0)	33.0	38.2 (11.4)	37.5	-7.9 (7.4)	-7.1	-5.3 (7.8)	-4.8	-6.7 (6.1)	-6.0	-4.6 (6.5)	-4.0
Visit 3	31.5 (9.6)	30.8	36.7 (10.9)	36.0	-10.3 (8.1)	-9.4	-6.8 (8.4)	-6.2	-9.1 (6.8)	-8.4	-6.1 (7.0)	-5.5
Hemoglobin (g/L)												
Screening	134.5 (11.2)	135.0	133.7 (10.9)	134.0								
Baseline	135.2 (10.8)	135.0	134.3 (10.5)	134.0								
Visit 1	137.8 (10.5)	138.0	136.0 (10.4)	136.0	3.3 (5.2)	3.0	2.3 (5.4)	2.0	2.6 (4.5)	2.5	1.7 (4.7)	1.5
Visit 2	139.6 (10.1)	140.0	137.4 (10.2)	137.5	5.1 (5.7)	4.8	3.7 (5.9)	3.5	4.4 (5.0)	4.2	3.1 (5.2)	3.0
Visit 3	141.2 (9.8)	141.0	138.9 (10.0)	139.0	6.7 (6.1)	6.5	5.2 (6.2)	5.0	6.0 (5.4)	5.8	4.6 (5.5)	4.5

Think about what just happened: We took a 20-row flat dataframe with 15 columns and created a professional two-page document with three-level headers, dynamic subtitles, parameter sections, and perfect formatting. **Without touching the source data**. Just declarative specifications.

Wait—How Did It Know the Column Widths?

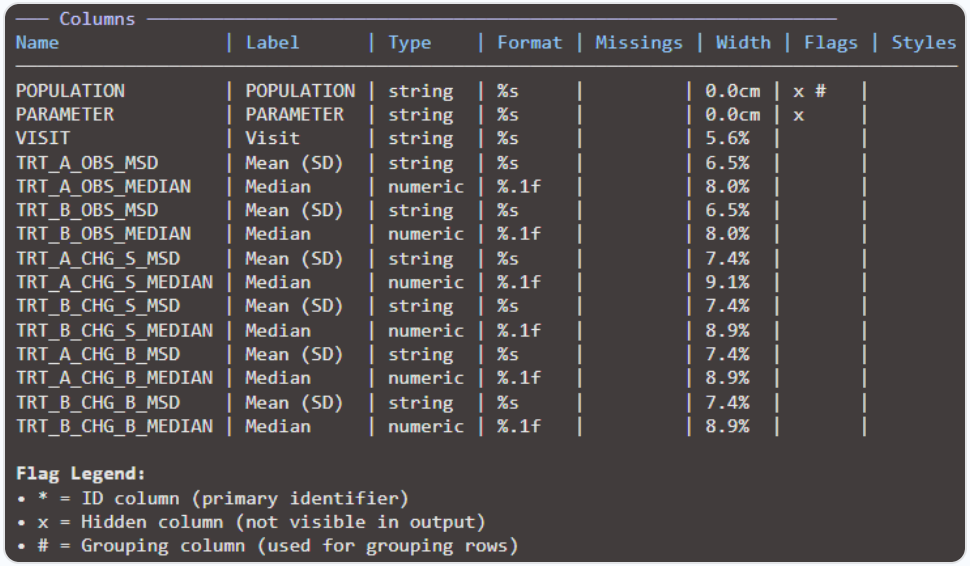
Sharp eyes might have noticed: We never specified column widths! With 15 columns to fit on a page, how did ksTFL decide how wide each should be? Let's peek under the hood:

We can see the column definitions by using `print()` on the spec object:

```
print(lab_spec)
```

This command prints useful information to the R console about all the definitions made to the specification.

You can see how the package calculated the column widths for each variable.



Columns

Name	Label	Type	Format	Missings	Width	Flags	Styles
POPULATION	POPULATION	string	%s		0.0cm	x #	
PARAMETER	PARAMETER	string	%s		0.0cm	x	
VISIT	Visit	string	%s		5.6%		
TRT_A_OBS_MSD	Mean (SD)	string	%s		6.5%		
TRT_A_OBS_MEDIAN	Median	numeric	%.1f		8.0%		
TRT_B_OBS_MSD	Mean (SD)	string	%s		6.5%		
TRT_B_OBS_MEDIAN	Median	numeric	%.1f		8.0%		
TRT_A_CHG_S_MSD	Mean (SD)	string	%s		7.4%		
TRT_A_CHG_S_MEDIAN	Median	numeric	%.1f		9.1%		
TRT_B_CHG_S_MSD	Mean (SD)	string	%s		7.4%		
TRT_B_CHG_S_MEDIAN	Median	numeric	%.1f		8.9%		
TRT_A_CHG_B_MSD	Mean (SD)	string	%s		7.4%		
TRT_A_CHG_B_MEDIAN	Median	numeric	%.1f		8.9%		
TRT_B_CHG_B_MSD	Mean (SD)	string	%s		7.4%		
TRT_B_CHG_B_MEDIAN	Median	numeric	%.1f		8.9%		

Flag Legend:
- * = ID column (primary identifier)
- x = Hidden column (not visible in output)
- # = Grouping column (used for grouping rows)

What you see: Each column's calculated width, label, visibility, and other properties.

Houston, We Have a (Fixable) Problem

Let's be honest: Automatic width calculation usually works great. But we just threw 15 columns at a landscape page. Physics has limits.

You probably noticed that in our table above **some of the values are split across lines**:

Screening	134.5 (11.2)	135.0	133.7 (10.9)
Baseline	135.2 (10.8)	135.0	134.3 (10.5)
Visit 1	137.8 (10.5)	138.0	136.0 (10.4)
Visit 2	139.6 (10.1)	140.0	137.4 (10.2)
Visit 3	141.2 (9.8)	141.0	138.9 (10.0)

Quick fix: Try to manually re-set widths in `define_cols()`.

Smarter solution: Look at the data. Do we really need separate columns for change-from-screening and change-from-baseline? What if we stack them vertically in the same cell?

The creative solution: Stack both change values vertically in one cell. Change-from-screening on top, change-from-baseline below. Cuts our column count from 15 to 11.

Can ksTFL do this? Absolutely. Remember, we're describing the output we want. Let's describe this layout...

The Clever Redesign: Stacking Change Values

First, hide the Baseline change columns—we're not deleting them, just hiding them. We'll use their values next:

```
lab_spec2 <- lab_spec |>
  define_cols(contains('CHG_B'), isVisible = FALSE)
```

Now let's combine text of the corresponding (paired) CHG values into a single value within one cell:

```
lab_spec2 <- lab_spec2 |>
  compute_cols(
    TRUE,  # On every row of the output
    c_glue(TRT_A_CHG_S_MSD, 'after', glue_col = TRT_A_CHG_B_MSD, separator = '<br>'),
    c_glue(TRT_A_CHG_S_MEDIAN, 'after', glue_col = TRT_A_CHG_B_MEDIAN, separator = '<br>'),
    c_glue(TRT_B_CHG_S_MSD, 'after', glue_col = TRT_B_CHG_B_MSD, separator = '<br>'),
    c_glue(TRT_B_CHG_S_MEDIAN, 'after', glue_col = TRT_B_CHG_B_MEDIAN, separator = '<br>')
  )
```

The `c_glue()` action: This concatenates two column values into one cell. The `<br>` separator creates a line break, stacking screening change above baseline change. Vertical space in our case is cheaper than horizontal!

📖 **c_glue() reference**

Update the Headers to Match

Since we combined the values, update the header text to reflect what's actually in those columns now:

```
lab_spec2 <- lab_spec2 |>
  add_span_header(4:7, 'Observed Values', stubOrder = 2) |>
  add_span_header(8:11, 'Change from Screening / Baseline', stubOrder = 2)
```

Add titles/subtitles—for this table, let's apply style to subtitle text to make it bold and centered as the primary table title. Also, let's add TOC marks:

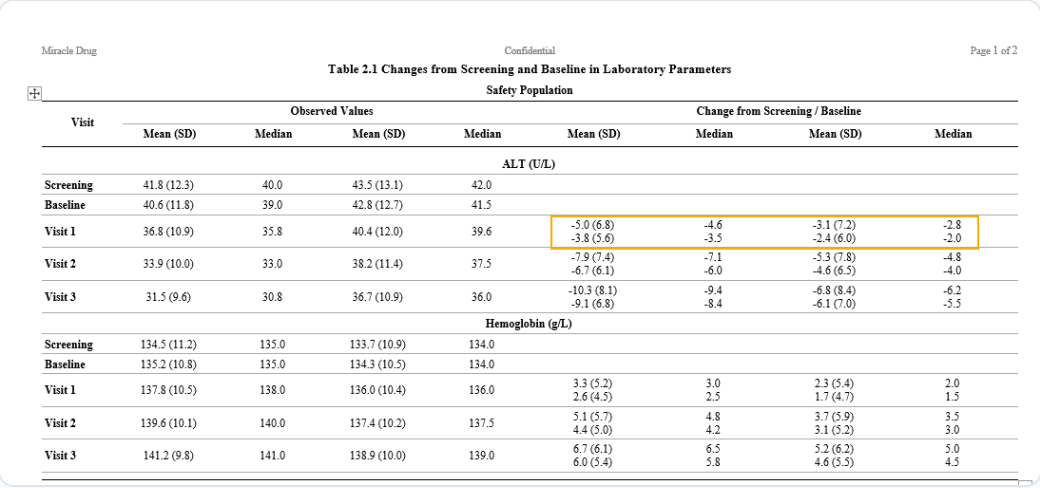
```
lab_spec2 <- lab_spec2 |>
  add_title('Table 2.1 Changes from Screening and Baseline in Laboratory Parameters',
            toclevel = 1) |>
  add_subtitle('#ByGroup1 Population', styleRef = f_combine('b', 'ac'), toclevel = 2)
```

Add parameter headers and some style enhancements:

```
lab_spec2 <- lab_spec2 |>
  compute_cols(firstOf(PARAMETER), c_addrow('above', value_from = PARAMETER, styleRef = f_combine('b', 'ac'))) |>
  define_cols(starts_with('TRT_'), valueStyleRef = 'ac') |>
  define_cols(VISIT, valueStyleRef = 'b')
```

The Improved Result

Much Better! Compare the Results



Miracle Drug Confidential Page 1 of 2

Table 2.1 Changes from Screening and Baseline in Laboratory Parameters

Safety Population

Visit	Observed Values				Change from Screening / Baseline			
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
ALT (U/L)								
Screening	41.8 (12.3)	40.0	43.5 (13.1)	42.0				
Baseline	40.6 (11.8)	39.0	42.8 (12.7)	41.5				
Visit 1	36.8 (10.9)	35.8	40.4 (12.0)	39.6	-5.0 (6.8) -3.8 (5.6)	-4.6 -3.5	-3.1 (7.2) -2.4 (6.0)	-2.8 -2.0
Visit 2	33.9 (10.0)	33.0	38.2 (11.4)	37.5	-7.9 (7.4) -6.7 (6.1)	-7.1 -6.0	-5.3 (7.8) -4.6 (6.5)	-4.8 -4.0
Visit 3	31.5 (9.6)	30.8	36.7 (10.9)	36.0	-10.3 (8.1) -9.1 (6.8)	-9.4 -8.4	-6.8 (8.4) -6.1 (7.0)	-6.2 -5.5
Hemoglobin (g/L)								
Screening	134.5 (11.2)	135.0	133.7 (10.9)	134.0				
Baseline	135.2 (10.8)	135.0	134.3 (10.5)	134.0				
Visit 1	137.8 (10.5)	138.0	136.0 (10.4)	136.0	3.3 (5.2) 2.6 (4.5)	3.0 2.5	2.3 (5.4) 1.7 (4.7)	2.0 1.5
Visit 2	139.6 (10.1)	140.0	137.4 (10.2)	137.5	5.1 (5.7) 4.4 (5.0)	4.8 4.2	3.7 (5.9) 3.1 (5.2)	3.5 3.0
Visit 3	141.2 (9.8)	141.0	138.9 (10.0)	139.0	6.7 (6.1) 6.0 (5.4)	6.5 5.8	5.2 (6.2) 4.6 (5.5)	5.0 4.5

What changed:

✓ Went from 15 to 11 visible columns
✓ Each CHG cell shows two values vertically
✓ Horizontal space saved
✓ No more wrapping issues
✓ Same data, different presentation

The breakthrough moment: Same source data, two different presentations. First version: 15 columns, wrapping issues. Second version: 11 columns, clean layout. **We changed the spec, not one byte of source data**. This is the ksTFL philosophy in action.

Bonus Feature: Automatic Table of Contents

Remember those `toclevel` parameters we added to titles and subtitles? They weren't just decoration. Watch what happens when we render with `toc = TRUE`:

```
report <- create_report(lab_spec2)
write_doc(report, '2_1_3_table', toc = TRUE)
```

What you get: A dedicated TOC page at the beginning of the document, automatically generated from the `toclevel` markers you placed in titles and subtitles.

… And Here Is the Final Result

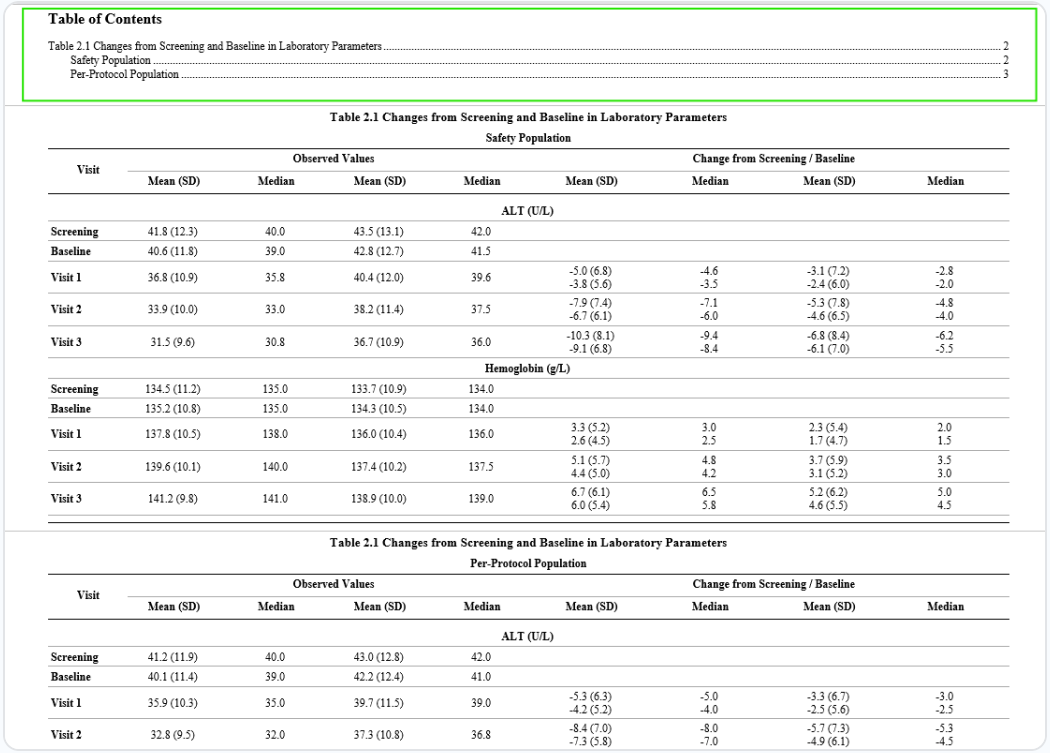


Table of Contents

Table 2.1 Changes from Screening and Baseline in Laboratory Parameters

Safety Population

Visit	Observed Values				Change from Screening / Baseline			
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
ALT (U/L)								
Screening	41.8 (12.3)	40.0	43.5 (13.1)	42.0				
Baseline	40.6 (11.8)	39.0	42.8 (12.7)	41.5				
Visit 1	36.8 (10.9)	35.8	40.4 (12.0)	39.6	-5.0 (6.8) -3.8 (5.6)	-4.6 -3.5	-3.1 (7.2) -2.4 (6.0)	-2.8 -2.0
Visit 2	33.9 (10.0)	33.0	38.2 (11.4)	37.5	-7.9 (7.4) -6.7 (6.1)	-7.1 -6.0	-5.3 (7.8) -4.6 (6.5)	-4.8 -4.0
Visit 3	31.5 (9.6)	30.8	36.7 (10.9)	36.0	-10.3 (8.1) -9.1 (6.8)	-9.4 -8.4	-6.8 (8.4) -6.1 (7.0)	-6.2 -5.5
Hemoglobin (g/L)								
Screening	134.5 (11.2)	135.0	133.7 (10.9)	134.0				
Baseline	135.2 (10.8)	135.0	134.3 (10.5)	134.0				
Visit 1	137.8 (10.5)	138.0	136.0 (10.4)	136.0	3.3 (5.2) 2.6 (4.5)	3.0 2.5	2.3 (5.4) 1.7 (4.7)	2.0 1.5
Visit 2	139.6 (10.1)	140.0	137.4 (10.2)	137.5	5.1 (5.7) 4.4 (5.0)	4.8 4.2	3.7 (5.9) 3.1 (5.2)	3.5 3.0
Visit 3	141.2 (9.8)	141.0	138.9 (10.0)	139.0	6.7 (6.1) 6.0 (5.4)	6.5 5.8	5.2 (6.2) 4.6 (5.5)	5.0 4.5

Table 2.1 Changes from Screening and Baseline in Laboratory Parameters

Per-Protocol Population

Visit	Observed Values				Change from Screening / Baseline			
	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median	Mean (SD)	Median
ALT (U/L)								
Screening	41.2 (11.9)	40.0	43.0 (12.8)	42.0				
Baseline	40.1 (11.4)	39.0	42.2 (12.4)	41.0				
Visit 1	35.9 (10.3)	35.0	39.7 (11.5)	39.0	-5.3 (6.3) -4.2 (5.2)	-5.0 -4.0	-3.3 (6.7) -2.5 (5.6)	-3.0 -2.5
Visit 2	32.8 (9.5)	32.0	37.3 (10.8)	36.8	-8.4 (7.0) -7.3 (5.8)	-8.0 -7.0	-5.7 (7.3) -4.9 (6.1)	-5.3 -4.5

Instant table of contents! Okay, for a single table it's overkill. But imagine a submission package with 50 tables and 20 figures. With `toclevel` markers on each spec, you get an automatic TOC with hierarchical organization and page numbers. **This is how ksTFL scales from one table to entire submission packages**.

📖 **Table of contents docs**

Get Started Today

Installation (r-universe recommended):

```
install.packages("ksTFL",
  repos = c(
    "https://crow16384.r-universe.dev")
    )

library(ksTFL)
```

Resources:

Full documentation
https://crow16384.github.io/ksTFL/

r-universe page
https://crow16384.r-universe.dev/ksTFL

Interactive tools:
`run_styles_editor()` — Visual style builder
`run_replay_app()` — Reproduce saved reports

Working Code Examples Repository
https://github.com/al-garik/ksTFL-examples

Found a bug? Have a question/suggestion?
https://github.com/crow16384/ksTFL/issues

Citation:

```
citation("ksTFL")
```

Aleschenkov I, Larchenko V (2026). *ksTFL: Framework for Clinical Tables, Figures, and Listings*. R package version 0.11.8, https://crow16384.github.io/ksTFL/https://crow16384.r-universe.dev/ksTFL