On tracing the `trip` test with `JSBox`

Doug McKenna

Abstract

A new TEX language interpreter library (currently called `JSBox`) I've been writing and debugging can accurately execute and completely trace the `trip` test, including recursive expansion, error interrupts, alignment table processing, and more. Relying on the log file that `JSBox` creates during tracing, this article explains how `JSBox` differs from TEX in tracing implementation and formatting philosophy, and reveals what's going on in a few of the many deliberately puzzling areas of the `trip` test.

Introduction

As is well-known in the TEX community, Knuth's self-described "diabolical" `trip` test helps validate any new or extended TEX language interpreter. Absent some similar test, the `trip` test is necessary — though not quite sufficient — to guarantee that one's interpreter is faithful to the core TEX language and the myriad lines of TEX code that have been written and relied upon for the last three decades. Knuth deliberately designed his test to be very difficult for both a non-conforming TEX interpreter and a human (conforming or not!) to understand.

The TEX code in `trip.tex` creates 16 pages of nothing useful typographically. But processing the convoluted input invokes nearly all the language's primitive commands, and most of the code paths each one depends upon. The test also relies on a font metric file, `trip.tfm`, with absurd ligature and kern programs in it. Many boundary conditions, where most errors occur, are deliberately triggered.

By design, there is almost no commenting nor documentation on what the `trip` test does. Validation generally means finding no non-trivial differences between an output log file and a reference log file. But this means that validation can occur without really understanding what the test does.

Worse, regardless of the job that TEX is performing, TEX only traces a portion of what's going on under its hood anyway. All of which is to say that the job of creating a conforming TEX language interpreter is not an easy or pretty one. Nonetheless, the `trip` test is invaluable in ferreting out bugs in any TEX language interpreter's implementation.

Tracing vs. hidden state

Basic user-interface theory teaches that modes foster human error. And all hidden state represents a mode of some kind. Hence, revealing hidden state — such as inserting individual temporary `print` statements in one's code — is always a key component of debugging. The TEX virtual machine and its quite complicated typesetting algorithms represent a great deal of hidden rules and state. So any TEX language interpreter must be pre-infused with special `print` statements to reveal (if asked) what is happening. The purpose of tracing is not only to create a record of what the macro interpreter has done on the user's behalf, but also to reveal what the machine is *not doing* that a user thinks it should be doing, because of some mode-induced user error.

To save code space, in one place (its inner execution dispatch loop) TEX generically traces the meaning of a primitive command that it is about to execute, on behalf of the upcoming snippet of code that implements that command. But because of this, there are still lots of holes in TEX's tracing that regularly cause user confusion. For instance, TEX only traces the first in a sequence of characters (i.e., a word), because characters after the first are handled by a separate inner loop looking for ligatures. Unfortunately, this violates the user's idea of the world, not to mention the reasonable expectation that if tracing is good enough for one character it ought to be good for them all.

TEX was designed to output quite short lines, and to break longer lines at arbitrary points, without regard to content. This in turn means there is no indentation to indicate in the log file where any subordinate set of executed commands start and end. All of which makes reading log files unpleasant. Regardless, often there is ensuing recursive expansion that is not (or only partially) traced.

Other examples abound: for instance, a large amount of complex behind-the-scenes processing occurs during any `\halign` or `\valign` command, yet much of it remains hidden during tracing. Many group contexts are unlabeled, e.g., when typesetting math formulas. `\global` definitions are not traced well, due to the internal design of how the prefix is processed. The places where a file is not found prior to being found are important to know when things go wrong. That missing information is a constant source of confusion ever-addressed by questions on various TEX-support web sites or mailing lists.

All of this hidden state represents a significant cognitive load on anyone reading a job's log file, because every lacuna violates the user's view of things, which is formed primarily by the sequence of commands and characters in his or her source code. There have been a variety of extensions made to TEX's tracing over the years, but as a relatively

recent user I find the results unsatisfactory: overly generic and/or still incomplete.

Redesigning tracing

As a self-defensive tech-support measure, I wanted my TEX language interpreter to be as communicative as possible to myself and any other user in explaining what it is doing, or not doing. So my goal has been to accomplish 100% tracing in a more complete and understandable format than what the classic TEX (or ε-TEX) engine does, without any significant hit on efficiency when not tracing. This means creating long-lined log files that no longer can be compared (e.g., with `diff`) to TEX's log files. The downside is that passing the ill-defined `trip` test becomes a tedious manual exercise, and either problematic or impossible, depending on what it means to be "the same". I don't really care, because I'm willing to label this new interpreter with something other than "TEX". The goal is to faithfully execute TEX source code and get the same typeset results.

In the `JSBox` library, each class of related primitives is implemented by a subroutine that is responsible for tracing each variant's operation, using a common set of tracing utilities and formatting rules. The utilities include a stack of output staging buffers in which to construct lines of text. The bottom buffer in the stack is always used for tracing. Higher buffer levels can interrupt tracing in the service of constructing strings, error messages, or other formatting. The stack is usually one level deep, and almost never more than two levels deep.

The start of any line of output from a program is usually better-defined in both time and space than the line's end. This means that it is the responsibility of any tracing or other output code initially to flush the last tracing line, then to start a new line, and to never worry about terminating that line. Different code paths can append to the line as needed, without being responsible for knowing the line needs to end. This is not dissimilar to what TEX does, but in `JSBox` we allow the end of the current text buffer to be trimmed prior to flushing. As we will see, interruptions can then be unambiguously formatted in a nice way. (It also allows `JSBox` to coalesce sequences of input characters so that all characters in a spacer-separated word are traced in one readable, delimited group of characters per trace, without doing any internal lookahead.)

Lines in a `JSBox` log file are not length-limited; the library can indent tracing to indicate different execution or nested group context stack levels. So every trace of a command or character(s) not only starts with a newline, but is then followed by indentation representing execution, group nesting depth, or trace-continuation status. (If occasionally indentation gets excessive, it is pinned.) For nearly all normal execution I find the clarity to be worth it. I place a high value on vertical alignment and white space. So a log file is best viewed in a fixed-width programmer's font.

The name, not the meaning, of the command being executed is enclosed by a pair of matching braces, as in `{\indent}`. A sequence of letter or other characters is also collected and placed at the start of the trace line, e.g., `{xyzzy}`, even though each input character is processed one at a time as it is read. If there is any further information or commentary about the command, a colon follows, and a description of what's going on, any special meaning, or what any collected argument value is, is appended next. Extra commentary is placed inside a pair of matching brackets. Unicode characters are presented as is (converted to UTF-8) and usually with added commentary showing both a base 10 and a hex integer value (and even more information for math characters). If a large amount of information is needed, any number of extra lines can be used in a single trace. All information on subsequent lines is indented to the same position as the line's initial information, *after* the announcement of the name of the primitive being executed or the word of characters being appended.

Internally, every new (multi-line) trace is assigned an integer code that immediately increments to guarantee uniqueness. If there is any chance that tracing might be interrupted by expansion, recursion, paragraph/page building output, an error message issued, or any sub-system tracing (such as a macro stack frame popping while looking for the next token), then the partial trace's text line has `"..."` appended. Later tracing then checks to see if there was an interruption. If not, the `"..."` at the end of the still-unflushed trace text is erased, and further tracing information is appended to the current line. But if there was any sub-tracing, error messages, or other output, then the trace buffer with its trailing ellipsis was flushed, and we create a repeated trace continuation line that *starts* with an ellipsis and re-traces the command again, at its usual indentation level, so that newly collected information can be presented to the user.

This is a lot of work, but clarity, not efficiency, is the user's focus during tracing. The TEX language's peculiarities make it really important to "go the distance" on this. Indeed, a significant portion of the `JSBox` library's code is devoted to tracing its own operation, in order to reveal hidden state.

Tracing examples from `trip.tex`

Consider the following line of nonsense code, from line 288 of `trip.tex`, but treated as line 1 here:

```
\raise1pt\hbox{\special{\the\hangafter} } \penalty-10000
```

This is a single `\raise` command, operating on a horizontal box with "embraced" contents, followed by a space, and then a `\penalty` command with its trailing integer argument. The TEX log file tracing this would contain (again, changing line 288 to 1):

```
{\raise}
{entering hbox group (level 2) at line 1}
{restricted horizontal mode: \special}
{blank space  }
{end-group character }}
{leaving hbox group (level 2) entered at line 1}
{horizontal mode: blank space  }
{\penalty}
```

This is concise, but unfortunately too concise. Important information remains confusingly hidden.

Here, on the other hand, is how `JSBox` traces the same code:

```
   {\raise}: by 1.0pt ...
      {\hbox}
>>> restricted horizontal mode
      {{}: entering \hbox group [level 2 at line 1]
         {\special} ...
            {\the} ...
               {\hangafter}: -12   [parameter]
        ... {\the}: Pushing {-12} [3 chars] onto input
     ... {\special}: {-12} [appending external command]
         { }: appending font \rip's inter-word glue [4.0 plus 2.0 minus 1.0]
      {}}: leaving \hbox group [level 2 at line 1]
.. {\raise}: appending hbox : [id=581] (0.0 + 0.0) x 4.0 [rigid] [2 items] shifted by -1.0pt [upward]
>> horizontal mode
   { }: appending font \rip's inter-word glue [4.0 plus 2.0 minus 1.0]
   {\penalty}: -10000 [always] [appending to horizontal list]
```

It's twice as many (longer) lines of tracing (and I've asked this journal's editor not to reformat the above to fit in a narrow column), which helps the reader discern pretty much everything that's gone on. So let's explain some of the design decisions that went into formatting the foregoing.

To start, the final (usually first) line of any trace that is interrupted ends in an ellipsis. And every trace so interrupted is re-traced (using the same indentation) after the interruption ends, showing the final information collected by the command's end. In the foregoing, the `\raise`, `\special`, and `\the` commands are interrupted and therefore re-traced using the latest state and argument information.

Every trace contains the command or a character or set of characters, enclosed in braces. The rule I try to follow is that, whatever the item is, it should be the same as what is in the user's original source code (TEX's tracing violates this in several ways).

Unlike TEX, we don't integrate changes to the layout's current typesetting mode as a modifier to the brace-enclosed item being executed or appended to the layout. Each such change is traced on its own line, with an indication (e.g., `">>>"`) of semantic nest stack depth, which is independent of execution stack or group context indentation. More importantly, this makes it easy to find, or easy to ignore, the state of the layout mode. And it doesn't violate the rule that the first and only thing executing is what's enclosed by a pair of braces at the start of each trace. That information is more important, and needed to synchronize source code with log file.

We strive to place a bracketed hint mentioning the internal meaning of numeric values. A "penalty" of `-10000` is really an (infinite) incentive to "always" do something. A negative `\raise` is upward on the page (unlike, say, PostScript, TEX's page coordinate system has its origin near the page top).

When a primitive appends a new item to the current layout list, it says so. For instance, when a spacer is processed in a horizontal mode, it appends a particular glue value, which is announced. And the `\raise` command, after all forward-looking recursive expansion is finished, is left with a horizontal box with two items inside that is appended to the current layout list after shifting. Another

hint reminds the user that this particular box cannot stretch or shrink. The library assigns a unique ID number to each box it constructs — searching a log file for a particular box is thus much easier.

There are other subtle formatting issues going on, in the service of maintaining vertical alignment of information. For example, the ellipsis in front of the re-traced `\raise` command is truncated by one dot, because there's one column too few in the indentation area in which to place three dots followed by a space (here, standard indentation is three columns per level).

Another difference from TEX is that nodes in a layout list (boxes, ligatures, kerns, penalties, glue, math, output nodes, etc.) are always described with a four-character identifier that never begins with a backslash. This regularizes vertical alignment of layout list dumps, which TEX minimally indents with visually noisy sequences of dots, sometimes followed by pseudo-commands (e.g., `\glue`) that don't occur in your source code. So the description of the box being appended by the `\raise` command is not `\hbox`. It is `hbox`, followed by a colon. The former is a command, the latter a type. Notice also that in talking to the user, we avoid the internal implementation and graph-theoretical term *node*. The generic word *item* suffices and is more user-friendly.

The `trip` test essentially consists of two parts. The first executes if one's virtual machine is uninitialized. The second runs if the interpreter is initialized from a format file, `trip.fmt`, created when the `\dump` command is issued at the end of the first part. The second, more substantive, part is where the `trip` test does most of its work. Because I desire this interpreter to be able to avoid using format files, `JSBox` currently treats a `\dump` as a no-op. Fortunately, by virtue of the trickery in `trip.tex` (see the definition of `\next` on line 90), this merely results in executing both parts of the `trip` test in one run.

In the first (format-creating) clause, `trip.tex` turns tracing off. Only various error or other messages are issued. Because `JSBox` can be compiled to suppress all "turn tracing off" commands that arise from source code, we can trace all parts of `trip.tex` without adding extra trace commands at the start of the input file.

On line 2 of `trip.tex`, the very first primitive is an `\immediate` command, followed by a `\catcode` command. The former's use is not an error, but is deliberately incorrect and/or unnecessary. That's because `\immediate` modifies only output-related commands that follow it (the `\catcode` command is not subject to the immediate vs. delayed execution distinction). An interpreter created by `JSBox` can be configured at run-time to comport with the constraints of a classic TEX82 interpreter, for which the `trip` test was designed. But `JSBox`'s client program can enable warnings for situations like this, because a command that does nothing when misused may still be worthy of the user's attention. So, the log file contains both the trace and the warning message:

```
   {\immediate}: [ignored]

Context : "trip.tex"[Line 2]

Warning : Ignoring \immediate. It only modifies
          output-related commands (e.g., \write),
          not \catcode.
Line 2  : \immediate\catcode `{ = 1 \endlinechar=13
          ^^^^^^^^^^
```

Trace lines are usually indented one level from the left margin, because most jobs start with the inclusion of a TEX source file or memory string. This pushes an input stack frame, to which trace indentation pays attention. While some might consider this a waste of space, it has the salutary effect of making non-trace messages (errors, warnings, etc.) easier to pick up in the log file, where they start at the left margin. And the extra space leaves room to signify re-tracing with a (partial) ellipsis, as our earlier example shows.

Because `\immediate` can be completely traced prior to the warning message being issued, there is no need for any trailing ellipsis: a re-trace wouldn't be able to add any new information. But in case warnings are disabled, the trace still provides a hint to the user that the command is useless and ignored.

Subsequent lines of interpreter-generated messages like the above are indented. This makes visual parsing of the log file much easier. Furthermore, the message itself is not generic. It has been tailored to include mention of the command (or character) which rendered the `\immediate` worthy of flagging. Non-generic messages are more work to create, but they keep the user grounded, preventing bad assumptions. Again, the goal in debugging is to reveal as much hidden state as possible.

Unlike TEX, the input line (or for longer lines, a portion thereof) is not broken into two pieces to implicitly show the position of the scanner when the message was issued. To do so violates the user's view of his or her source code, and thereby unnecessarily adds to a cognitive load at an inopportune time. So `JSBox`'s scanner maintains the starting and ending position of each item parsed on an input line, and preserves that information for the benefit of any

formal message reporters. In the case of writing the error to a fixed-width format log file, we underscore — as best we can in a fixed-width log file font — the command (or character) responsible for the message. This is sometimes difficult to do in a manner that doesn't confuse the user, even though it might be internally accurate. Non-generic error messages can alleviate the problem somewhat.

Here is another traced snippet from line 4 at the start of `trip.tex` that illustrates more of `JSBox`'s tracing philosophy and formatting. This code temporarily changes the category code of the math formula shift character (`$`) inside a group context.

```
{\catcode}: ‘$ <- 3 [math shift] [no change]
{{}: entering simple group [level 1 at line 4]
   {\catcode}: ‘$ <- 13 [active] [was 3 = math shift]
{}}: leaving simple group [level 1 at line 4]
     restoring [mapping] \catcode of ‘$ to 3 [math shift]
{ }: [ignored in vertical mode]
```

In this case, it is now TEX that would trace the above using nearly twice as many (shorter) lines:

```
{\catcode}
{reassigning \catcode36=3}
{begin-group character {}
{entering simple group (level 2) at line 4}
{\catcode}
{changing \catcode36=3}
{into \catcode36=13}
{end-group character }}
{restoring \catcode36=3}
{leaving simple group (level 2) entered at line 4}
{blank space  }
```

I invariably can't recall the implicit (hidden) meanings of numeric codes, such as `36` or `3`. So when `JSBox` traces the `\catcode` command, adding commentary (i.e., `[math shift]`) on the syntactic meaning of the numerical argument `3` saves mental energy and prevents errors, and the character itself is used, not just its ASCII code `36`. We also note actual value changes, but unlike TEX, we don't use two more lines to accomplish this trace generically. And in the main trace we strive to inform the user of the new value *prior* to what the old value was, because the new value is what the user is nearly always interested in.

`JSBox` internally traffics in full 21-bit Unicode code point integer values, with all of them above the initial ASCII range initially classified as characters of type "other". And any Unicode character (code point) can have a syntactic catcode assigned to it. For printable ASCII characters in the initial 7-bit plane, such as the `$` above, we don't output the character's integer code (for arbitrary Unicode, we would). In extended mode (i.e., when not limited to just TEX82 features) the `JSBox` library can handle non-UTF-8 Unicode using `^^u`*xxxx* or `^^U`*xxxxxx* extended escape sequences to specify Unicode code points in hex, or via the usual `\char` command, which will take any 21-bit integer argument (but limited to 8-bit values in TEX82 emulation mode).

Notice also that the information about popping a group context stack frame, and restoring any non-globally changed values, is properly traced solely by the recognition of the `}`. Restoration is indented to indicate its subordinate status to the closing brace, and there can be an arbitrarily long list of lines announcing each restoration. Because it's hard to distinguish between various control sequence names, a hint as to whether a name is a parameter, register, mapping, etc., is also inserted for good measure.

Finally, by announcing that a space in vertical mode is ignored, there is no need to label the space character as a blank space; `{ }:` does a perfectly fine and unambiguous job.

Here is another example, from line 10 of `trip.tex`:

```
\defaulthyphenchar=‘-
```

`JSBox` traces this as

```
{\defaulthyphenchar}: - [was ^^@] [‘- = "2D = 45]
```

The new character code for the parameter is shown, followed by bracketed commentary on the parameter's previous value, which changed. Then further commentary on the integer and hex value of the new character is added in case it might be useful. Notice that the old value was 0, a null. Unlike TEX, `JSBox` strives never to write a null byte to a log file or to the terminal. Programs that display text files do not treat nulls uniformly. So unless it's writing a data file, `JSBox` converts each null byte to a printable `^^@`.

The general philosophy I'm guided by is that too much information is a lesser evil than too little. For commands that expect character code point values, the numbers are there, but on the right, in commentary, where it can be easily ignored.

Consider lines 59–60 of `trip.tex`:

```
\def\weird#1{\csname\expandafter\gobble\string#1 \string\csname\endcsname}
\message{\the\output\weird\one on line \the\inputlineno}
```

ε-TEX's trace of this peculiar code would be

```
{\def}
{changing \weird=undefined}
{into \weird=macro:#1->\csname \expandafter \gobble \ETC.}
{blank space  }
{\message}

\weird #1->\csname \expandafter \gobble \string #1 \string \csname \endcsname
#1<-\one
{\csname}
{\expandafter}
{\string}

\gobble #1->
#1<-\
{\string}
{changing \one \csname=undefined}
{into \one \csname=\relax}
\one \csname on line 60
{blank space  }
```

whereas JSBox can trace the same input as follows:

```
   {\def}: \weird#1->\csname \expandafter \gobble \string #1 \string
                     \csname \endcsname
   { }: [ignored in vertical mode]
   {\message} ...
      {\the} ...
         {\output}: [parameter] ->
  ... {\the}: Pushing {} [token list] onto input

      Calling \weird #1->\csname \expandafter \gobble \string #1 \string
                         \csname \endcsname
              #1: \one
         {\csname} ...
            {\expandafter}: postponing {\gobble} until after expanding {\string}
            {\string}: Pushing {\one} [4 chars] onto input

            Calling \gobble #1->
                    #1: \
            Returning from \gobble  [empty body]
            {\string}: Pushing {\csname} [7 chars] onto input
     ... {\csname}: 11 characters collected
         {\endcsname}: \one \csname constructed [=\relax [internal]] [{\one \csname} has a space in it]
      Returning from \weird, resuming reading from file "trip.tex"
      {\the} ...
         {\inputlineno}: 60
  ... {\the}: Pushing {\one \csname on line 60} [23 chars] onto input
.. {\message}: [to "trip.log" and terminal]

\one \csname on line 60

   { }: [ignored in vertical mode]
```

In lines 1–2, we wrap a longer token list onto a new line without breaking any command name internally, and we continue the token list indented to the same column as it started on, just to the right of the `->`. A longer list will be truncated, but `JSBox`'s threshold is larger than TEX's.

Like TEX, we insert a blank line in front of each macro call, but unlike TEX, we label each macro call with `Calling` to distinguish macros from primitives (or `\let`-created synonym names). Arguments collected are indented further — their collection is all part of the same trace of the macro call. We additionally can trace the end of the macro, when its stack frame gets popped, and announce later (at line 22) where the next input will come from (either a file, or a previously called macro in which execution was nested). In between, we've entered a new nested and indented command execution level.

At line 13, the often confusing `\expandafter` command is traced non-generically. At line 18, the `\gobble` macro expands to nothing, so we add a commentary hint saying `[empty body]`. At line 21, we specifically add commentary for any constructed control sequence name that (here deliberately) has a space in it, which can otherwise be very confusing.

Finally, we place a blank line on either side of any non-tracing text being output to the same destination (log file or terminal) that interleaves with traces. This sets the output off and makes it much easier to find by scanning down the page, especially as most such non-tracing, internally generated message output is not indented. (`JSBox` can insert an extra blank line automatically between any two different classes of text in log/terminal output.)

Redesigning the `\show...` commands

Each of the TEX language's `\show...` commands formats and prints the value of some internal variable or list. But TEX re-uses some of its error reporting machinery to do so, which I find confusing. For example, lines 29–30 of `trip.tex` seem to me to result in a hard-to-read formatting mess:

```
> \errorstopmode=\errorstopmode.
l.29 ...=256 \show\errorstopmode

> \rip .
<recently read> \font


l.30   \showthe\font
                     \showthe\pageshrink \showthe\pagegoal
> 0.0pt.
l.30 ...font \showthe\pageshrink
                                 \showthe\pagegoal
```

There's too much extraneous information, and it's a cognitive load to be presented with the effect of the command *prior* to seeing the command causing that effect. So `JSBox` avoids displaying non-relevant parts of the input line and/or breaking it apart to show where the scanner is, puts cause and effect back in order, skips the unnecessary detail about what was `<recently read>`, and displays just the answer on its own line. Also, we label lines with `Line`, not `l.`, because of the time-honored principle that a lowercase `l` in a fixed-width font will invariably be confused with the digit `1`. A new trace of lines 29–30 "shows" the difference:

```
      {\show}: \errorstopmode

Line 29 | \show \errorstopmode
\errorstopmode

      {\showthe} ...
         {\font}: \rip
  ... {\showthe}:

Line 30 | \showthe \font
\rip

      {\showthe} ...
         {\pageshrink}: 0.0pt
  ... {\showthe}:

Line 30 | \showthe \pageshrink
0.0pt
```

Without interleaved tracing, this would simply be:

```
Line 29 | \show \errorstopmode
\errorstopmode
Line 30 | \showthe \font
\rip
Line 30 | \showthe \pageshrink
0.0pt
```

The delimiters `>` and `.` are not used, because I find that they add more ambiguity/noise to TEX's output than they resolve. Also, `JSBox` doesn't insert blank lines between these similar `\show...` commands because it knows that each result fits on one line. For other `\show...` commands that result in multiline answers, such as `\showlists` or the extended `JSBox` `\showfont` command (which shows the metrics and all other data of an entire loaded font), blank lines are used to help the user understand where the command's group of output lines ends.

Tracing alignments

Perhaps the most complicated primitive commands in the TEX language are `\halign` and `\valign`, each of which converts a one-dimensional stream of commands and text into a two-dimensional table on the page. Both commands work almost identically, by swapping horizontal rows with vertical columns. And they're recursive, since an element of a table's cell can be a sub-table. Material that looks like it might be executed is recorded, and material that looks like it might be recorded is executed. Expansion can occur. There are hidden contexts. The purpose is to allow the entire power of TEX to be applied to any cell in a table. Knuth describes them as working almost magically.

Consider line 120 of `trip.tex`. It contains a curious empty table, as part of the `\output` routine for the page executed later at line 150:

```
\globaldefs1\halign{#\tabskip\lineskip\cr}
```

Among other things, this tests whether one has executed a `\tabskip` command, with attendant expansion and implicit global definition, in the alignment's preamble, rather than recording it into the preamble's token list(s), as would be nearly all other commands and characters. TEX82's trace of this, as taken from `trip.log`, is about as minimal as can be:

```
{\globaldefs}
{\halign}
```

ε-TEX does a little better with its tracing extensions turned fully on, but here's what's really going on under the hood, as traced by `JSBox`, using its interruption and indentation rules:

```
1     {\globaldefs}: 1 [was 0]
2     {\halign}: building successive rows, each containing entries in horizontally tabbed columns ...
3        {{}: entering \halign group [level 3 at line 151]
4  ... {\halign}: [preamble] recording templates for each column ...
5           {#}: end of prefix material for column 1's template; collecting suffix
6           {\tabskip}: [not recorded in template] ...
7              {\lineskip}: 0.0pt plus 40.0pt   [parameter]
8       ... {\tabskip}: changing \tabskip [inter-column glue] to 0.0 plus 40.0 [global]
9           {\cr}: end of column 1's suffix and template
10 ... {\halign}: preamble has declared template for 1 column ...
11                \tabskip [= 0.0]
12                column 1:        [no extra material to insert]
13                \tabskip = 0.0 plus 40.0
14          {column entry}: entering hidden alignment item group [level 4 at line 151]
15          {column entry}: leaving hidden alignment item group [level 4 at line 151]
16       {}}: leaving \halign group [level 3 at line 151]
17 ... {\halign}: [done] appending 0 rows of aligned material
```

As `\halign` fires up (see line 2 above), exactly what is about to happen is announced, because there's simply not enough information in the name of the command to disambiguate what the command does, should the user not be sure. As processing of input proceeds, the `\halign` will retrace itself three more times (with ellipses as appropriate), at the same indentation level, even though the execution stack level is changing up and down at the same time. Notice that every token in the source code is represented as the start of a full trace line or lines — tracing should not break the user's mental model of what's going on. Because there will be one last trace line announcing the final result of the command, we indent the opening `{` and closing `}` of the `\halign` group, to make it easier to scan down the log file looking for the start of the final trace line.

At line 4, the alignment's preamble starts being recorded. When the preamble material ends with the first `\cr`, we trace again, and then synopsize the template material for each column. Unlike TEX's opaque and overly mathematical *u*-part and *v*-part terminology, I've used the more descriptive and user-friendly terms *prefix* and *suffix*.

TEX processes each column's material inside a hidden group context, as if surrounded by a pair of braces, to prevent changes to registers and parameters from leaking to the following column's material. In this example, there is no material at all, so after all is done and the final `}` is "executed", the column is empty, leaving the table with no rows at all. If a sub-table is processed, the alignment stack's current level (other than 1) is also traced, which makes searching for matching traces easier.

If that's what goes on in an empty table, imagine how important full tracing is in a non-trivial example, of which there are many more in `trip.tex`.

Tracing synonyms and conditionals

TEX traces the *meaning* of the executing control sequence, not its name. So if the control sequence is a synonym for a register, or a primitive command, or other name (e.g., created by a `\let`), it can get confusing. For example, here's line 161 from `trip.tex`:

```
\dimendef\varunit=222\varunit=+1,001\ifdim.5\mag>0cc0\fi1pt
```

This defines a name `\varunit` for the dimension register at index `222`, and assigns a value to it. The value's digits are conditionally expanded using an `\ifdim` conditional smack dab in the middle of collecting the value's digits. The test asks if half of `\mag`, whose value is `2000`, is greater than `0` (Cicero) points, which is true. But there's an implicit conversion from an integer `.5\mag` (`1000`) to a fixed-point dimension (`0.01526pt`).

ε-TEX would trace some of the statements of this line generically as

```
{\dimendef}
{changing \varunit=undefined}
{into \varunit=\relax}
{changing \varunit=\relax}
{into \varunit=\dimen222}
{\dimen222}
{\ifdim: (level 1) entered on line 161}
{true}
{\fi: \ifdim (level 1) entered on line 161}
{changing \dimen222=0.0pt}
{into \dimen222=1.001pt}
```

whereas `JSBox` traces the same code, in both shorter and more faithful-to-the-source fashion, as

```
   {\dimendef}: \varunit = \dimen222
   {\varunit} ...
      {\ifdim} ...
         {\mag}: 2000 = "07D0   [parameter]
  ... {\ifdim}: 0.01526 > 0.0 ? true [line 161]
      {\fi}: [end of \ifdim on line 161]
.. {\varunit}: \dimen222 = 1.001pt [was 0.0pt]
```

Notice that the extra definition to an intermediate `\relax` is an arcane internal TEX implementation detail (see the comments in `"tex.web"`) that is of no interest to 99.999% of users, but ε-TEX traces it anyway due to the generic nature of how it reports changes to interpreter values. `JSBox`'s focused tracing during name definition elides this extra internal reference management step.

We announce `{\varunit}` at the start of its trace (as opposed to `{\dimen222}`), because it comports with the source code. When the command has finished recursively expanding its argument, the final re-trace shows the value assigned, and the register index, and the former value, all in one line.

When `JSBox` traces a conditional, it shows the values used in the test as such, followed by a `?`, followed by the answer, either `true` or `false`, all in one line. A hint can be added, followed by the test's line number as final commentary. If a `\fi` ends a multi-line conditional, the line range is used.

The only other information that might arguably be brought to the user's attention is what happens to the single character `0` collected as another digit in the dimension's value, and the fact that the original source tries to present (after the conditional is through mucking with things) the value `1.00101pt` as `\varunit`'s value. But the final `1` is insignificant as there is round-off to `1.001pt`, which is what's traced.

Line 360 of `trip.tex` has a crazy `\ifcase` conditional statement, with nested `\ifcase`s:

```
\ifcase\iftrue-1a\else\fi
\ifcase0\fi\else\ifcase5\fi\fi
```

that `JSBox` traces fairly cogently and concisely as

```
   {\ifcase} ...
      {\iftrue}: true [line 360]
.. {\ifcase}: looking for case -1 [only matches
              an \else clause, if any] [line 360]
      {\else}: false
      {\fi}: [end of \iftrue on line 360]
      {\ifcase}: looking for case 0 [matched]
                 [line 360]
         {\relax [internal]}
      {\fi}: [end of \ifcase on line 360]
   {\else}: true [no previous case matched -1]
      {\ifcase}: looking for case 5 [line 360]
      {\fi}: [end of \ifcase on line 360]
   {\fi}: [end of \ifcase on line 360]
          [no case matched]
```

A TEX language interpreter must insert an internal `\relax` to enable expansion to parse an inner nested conditional involved, as here, in constructing the condition of an outer conditional. The only remaining information that might be traced in some way, but isn't, is which commands or characters in the source code are skipped over for false matches, and why. This would help explain the mysterious disappearance of the letter '`a`' when executing the foregoing code (it ends up being part of the outer case `0`, which is skipped while scanning for case `-1`).

The `trip` test has another strange boundary condition test: what happens to an `\if...` statement with more than one `\else` clause. Peculiarly, sometimes it's an error, and sometimes it's not. So

I had fun ensuring that JSBox's trace of line 389, where multiple \elses are not an error,

```
\ifx T\span\else\par\if\span\else\else\else\fi\fi
```

would be enlightening to the user:

```
   {\ifx}: T = \span ? [Chr = Cmd] false [line 389]
   {\else}: true
      {\par}: [building more page]

% t=30.0 plus 42.0 plus 1.0fil minus 8.0    g=16383.99998 b=0 p=0 c=0#    [# = best break so far]

      {\if}: \span=\relax [internal] ? [neither is a character, treated as equal] true [line 389]
      {\else}: false
      {\else}: still false [ignoring extra \else clause]
      {\else}: still false [ignoring extra \else clause]
      {\fi}: [end of \if on line 389]
   {\fi}: [end of \ifx on line 389]
```

Note that when tracing an \else clause, its line number is suppressed if it is the same as that of its initial \if... condition. This makes tracing short conditionals like the above less noisy. And again, notice in the above trace the paragraph building data that's caused by executing the \par command. When output of a different class is created, we strive to set it off with blank lines from the surrounding indented trace lines, so as to stand out as non-tracing.

Other tracing niceties

Every hundred or so traces, JSBox inserts a short context announcement that shows the current set of included files and the line number in each from which the scanner is reading. For example, if we were including trip.tex from another file, such as test.tex, the announcement might look like:

```
>> display math mode
      {^}: [superscript]
      {\mathop}

  "test.tex"[Line 4] > "trip.tex"[272]

      {b}: letter maps to "7162
           [class 7 (inner);
           \fam 1 \char "62 =  98 = b]
      {\nolimits}
```

This makes it easier for the reader to understand where nearby tracing in the log file is arising from in an included source file. It also lessens the need to include line numbers in individual traces.

Formal error reports are usually preceded by a full execution stack dump, showing included files, the macro call stack, and an indication of empty stack frames, deleted to allow tail recursion.

When executing trip.tex the execution stack is never very deep when errors are reported. So the following is an example of a stack dump showing nested macro calls while executing a test file called plainstory.tex that relies on footnote macros defined in the plain format, whose non-\dumped source code is read in at the start of the job:

```
Context | "plainstory.tex"[Line 112] >> \story >
        | \rhubarb > \fubaru > \bar > \foo >
        | \testparagraph > \note >> \footnote >
        | \vfootnote
```

The >> indicates that there were empty stack frames that were deleted during regular stack cleanup (the last two macro names are defined in plain.tex).

Another nicety in JSBox, tracing the resolution of input files, is not as demonstrable using the trip test. JSBox is a system-agnostic library that can be linked into a client program. The client is responsible for mediating between the interpreter and the system, and must install a callback function with which the interpreter asks the client to do various tasks.

In particular, when JSBox executes the \input command, it first asks the client to vet each character in the file name. This lets the interpreter issue an error message at precisely the right time for any illegal or unwanted character. Once the file name is collected and analyzed a bit more, the interpreter then asks the client to construct a list of directories in the client's file system where to look for that file. As the list is iterated, a back and forth between the client and the interpreter allows all the unsuccessful folders to be traced as well as the final one where the file is found. To gate this, JSBox implements another integer parameter, \tracingfiles, that can

be set to `1`. For example, when a source file inputs `plain.tex`, the trace might look like:

```
{\tracingfiles}: 1 [was 0]
{\input}: plain.tex
          [not found at ./Projects/TUG Article/plain.tex]
          [not found at ./JSBox/Projects/plain.tex]
          [aha! it's at ./JSBox/Library/Formats/plain.tex]
```

Thus, the input resolution strategy is up to each individual client program, but the interpreter can reveal and record some of that strategy for the user at the right time and in the right place. This is particularly important when, contrary to the user's expectation, no file is found, or when an incorrect file of the same name but with a different path is found first (e.g., the wrong version of a file).

The complete JSBox trip test trace

The foregoing should give the seasoned TEX log file tracer/reader a good idea of how the execution of a piece of TEX source code can be traced in a much nicer and more useful manner than what the TEX engine does, even considering the ε-TEX extensions.

The entire and latest complete `JSBox` trace of the `trip` test can be found as a PDF at `http://www.mathemaesthetics.com/JSBox/triplog.pdf`. It's about 160 landscape pages long. I continue to refine and add to the library's tracing, so the log file will be updated occasionally. Indeed, I tweaked at least one feature (eliding line numbers in `\else` traces when on the same line as the initial `\if...`) as a result of creating examples for this article.

Conclusion

Over the course of this multi-year project, I have perhaps four times re-designed and re-written how tracing should work, each time realizing that what I was doing in the previous iteration wasn't complete enough. There are still a few indentation bugs in the code, and some older cruft that deserves to be re-written or cleaned up.

The `JSBox` library is a work-in-progress. Over 2000 pages of portable C code, half of it English comments, it supports plenty of other interesting features, a description of which can be saved for another time. The library is not yet ready to be deployed as a new TEX language engine. When I started on this project in 2009, I had very little idea of the complexity of the task of being compatible with the TEX language, which I barely knew. Its many quirks as a Turing-complete macro language seem as closely tied to its program's implementation details as they are to any overarching language syntax specification. And the greater TEX ecosystem is even more complex. Indeed, my goal of complete tracing is to make executing TEX code as self-documenting as possible.

Debugging the interpreter so that it would execute the `trip` test correctly — i.e., mathematically and functionally equivalent to what TEX does — required many months of work. Each bug it revealed had to be fixed prior to moving on to the next, due to the cascading nature of layout calculations. In several cases, I had to completely re-implement how certain primitives worked internally, after my initial assumptions proved invalid.

There's a lot of work left to do. But at the very least, this interpreter can — in full and gory detail — now tell the world nearly all of what it's doing under the hood. Which is why the chapter on tracing utilities in `JSBox`'s source code has the title

```
Understanding Interpreter Execution
Vanishes Without A Trace!
```

◇ Doug McKenna
Mathemæsthetics, Inc.
PO Box 298
Boulder, Colorado 80306, USA
doug at either mathemaesthetics
dot com or dmck dot us

Editor's note: An overview of `JSBox` is available via the slides at `http://tug.org/tug2014/slides/mckenna-JSBox.pdf`. We hope to publish additional related papers on this work as it progresses.