

Flagging

Hans Hagen

1 Introduction

In this summary I will describe how one goes from using color clues in a PDF presentation at BachoTeX to an equally colorful signaling feature in ConTeXt processing. The presentation was about flags. For quite a while flags have been used for communication, on sea and on land. Before signals became digital, a system of widespread semaphores on towers and hills were using a system of flags that essentially encoded messages over a distance. Today's communication technologies that rely on inventions and concepts of those times, like encoding and encryption, owe a debt to this.

When discussing possible topics for BachoTeX, which this year focused on educational usage, we wondered if we should spend some words on accessibility and color as part of that discussion. Bringing a child to the meeting also means coming up with a presentation. So, when Tove Sundqvist suggested a quiz with flags, a colorful presentation had to be made. And implementing the visuals for that inspired us to think a bit about flagging progress, success and/or failure of a TeX run using colors. After all, we had the technology ready.

So, we backtrack a little in time and start with signals, make an excursion to flagged PDF, where we talk colors, then come back to signaling using colors, and finish with explaining how this integrates in TeX. It is also an illustration of how getting side tracked can lead to an unexpected new ConTeXt feature.

2 Modern error reporting

We start with a short excursion into TeX, a tool that we assume the reader is familiar with. Say that we have this document source:

```
\starttext
Hello
\stoptext
```

This is about as minimal as a document encoded in ConTeXt can be. When in the editor you hit the process button (assuming there is one, like F12 in SciTE), ConTeXt will process that code into a document. But what will happen with the following?

```
\mainlanguage[swedish]
\starttext
Hej (aka \Hello)
\stoptext
```

You will get an error message, like this

```
<line 3.2>
Hej (aka \Hello)
)
```

Hans Hagen

doi.org/10.47397/tb/46-3/tb144hagen-flagging



Figure 1: A QR-coded pop-up error message.

The control sequence at the end of the top line of your error message was never `\def`'ed. You can just continue as I'll forget about whatever was undefined.

That is not a message the youth will get thrilled about. They more likely expect something like that shown in figure 1.

And indeed, we can produce this! We either run ConTeXt with

```
context beyond-tagging --signal=qr
```

or add this at the top of the file:

```
% signal=qr
```

Then the user can take a phone and scan the code to see what really happened. And, because humans are good at fast visual recognition, after a while this message will be recognized faster than one can read!

In order to test how users respond to this kind of feedback, Willi Egger made a set of ear tags that we put various TeX and MetaPost error messages on. We can't make that as small as normal (goat) tags (figure 2) but wood is kind of neat anyway. Here 'tagged' is an intended pun: instead of tagging the PDF output we tag TeX errors. Those we then tested on the attendees of the meeting who could use their phones to check what the message said. Yes, TeX error messages can be confusing but also humorous. And there are examples that some users will never have seen. The starter set is shown in figure 3.



Figure 2: Real eartags (for goats).



Figure 3: Left: blank wooden tags; right: tags with QR codes

When we demonstrated this, those present could indeed easily translate the shown codes into familiar text. The number of error messages in $\text{T}_{\text{E}}\text{X}$ and MetaPost is not that large. We collected some 30 from the source code on those wooden tags so that those interested could take one, but left out the ones that get assembled. Of course assembled messages will also be reported.

This `signal` feature is a plug-in mechanism. Errors are already intercepted by a callback which means that the engine itself doesn't report them directly. This provides us a way to give a bit more detailed feedback. A plug-in like `qr` goes even further and spawns another job generating the error document showing the message as a QR code. You can imagine other plug-ins and indeed we'll come back to that later.

3 The flagging experiment

When the team prepared this flagged PDF talk, we were also reminded of the fact that one of the main complaints of the PDF accessibility validator promoted at (Con $\text{T}_{\text{E}}\text{X}$ t colleague) Mikael Sundqvist's university department was about distinctive colors. One problem there is that when color is used only to make a document more attractive without it having meaning, a checker can't know that. Flags use bright colors so in the presentation we had to settle for that.

And so we wondered if colors could be used as signals. As $\text{T}_{\text{E}}\text{X}$ is embedded in the scientific world, we first wanted to test if indeed color was a problem for $\text{T}_{\text{E}}\text{X}$ users. It was hard to come up with a good test, but luckily for us, Mikael's daughter Tove Sundqvist came with the idea to test the audience on flags and she was willing to participate in an experiment with flagged PDF files. One can only imagine what happens in a critical situation when the flags of two countries can't be seen as different due to how colors are perceived. This being a conference, of course we had to formulate a few hypotheses:



Figure 4: The test setup. Flags shown are Lebanon (left) and Portugal (right).

How well can a user distinguish colors (here in a PDF file) or: do we really have to be as dull as the university checker wants us to be? As flags use rather primary colors, can we stick to these?

and:

What kind of signaling works best when we have an error? It makes little sense to pop up a colorful PDF file. We can't really test this at a meeting like this but at least we can form an impression.

The experimental setup involved, as usual at $\text{BachO}_{\text{T}_{\text{E}}\text{X}}$, a beamed presentation but this time a little bit of multi-media was added. Tove showed a flag and asked the audience what country it was from. As a double check the right answer had to be complemented with a loud 'Hello' in the related language! Showing the page with a flag also triggered up to four colored light bulbs which permitted us to observe if that visual clue was helpful. We will reveal how that was done later. In figure 4 we see the test setup using a large monitor with the four lights at the sides (we had to test and prepare this via Zoom beforehand).

One can wonder if in an audience with mostly males¹ there will be complaints about color usage, given that color blindness is kind of omni-present the room should have had two candidates present. We never noticed. What we did notice is that the rather international audience recognized some 50 percent of the flags, but not because of the lack of color. It is more that number of distinctive colors used is small, so often ornamental additions matter more. This might indicate that (as with serifs in glyphs) colorful abstractions can benefit from some additional distinctive additions.

In the device we will discuss later we have built in a few alternative color palettes. An interesting

¹ According to Wikipedia up to 1 in 12 males (8%) and 1 in 200 females (0.5%) have color deficiencies.

observation is that these look okay on paper, but translated to light bulbs they are less distinctive, so we will not elaborate on this here. We might come back to it at the upcoming ConTeXt meeting. In this perspective one can think of flags with washed out colors due to long exposure to sunlight.²

Back to the presentation. Once the country was known, coming up with the right ‘hello’ was less of a problem although there was some disagreement and discussion among the Czech and Slovenian participants about this and with Tomas Hala being in charge of additional translations, maybe future versions of ConTeXt will have proper hello labels.

So what can we conclude? Maybe only that primary colors when seen next to each other on the average are okay as visual clues, so we can move on to using color for signaling.

4 Using colors as signals

The (preparation of this) experiment was very promising (and fun) which made us decide to come up with a variant on the QR code TeX error. Not only did we think of errors, but because these color bulbs were mesmerizing it also made sense to use them for showing the state TeX is in. So, in addition to the PDF triggering colors we also demonstrated run states, now also known as ‘signals’.

When a run is started a blue light is shown; when the run is successful that one turns yellow. If an additional run is needed the second light becomes blue. Eventually we’re finished and the current (or all) light(s) is then either red (fatal error), orange (too many runs needed), or green (we’re fine).³

Already at BachoTeX we wondered about a follow up. When we picked up Mojca Miklavc on the way to the meeting, she gave us an LED strip that we could play with in the evenings as I had a ZigBee LED controller with me. However, on the way back she suggested using a dedicated solution and when we dropped her off on the way back, we left with an RPI Pico that Vincent had prepared for us to play with. And that is where it really took off.

In the weeks following BachoTeX, after experimenting with the color strip, we moved on to a 7cm diameter circle with 24 addressable WS2812B RGB LEDs. After first playing with WiFi we started playing with the serial connection of the RPI Pico. We will not go into all the details; for that we have a manual. Here we stick to the main outcomes: a device that can be used to keep track of progress

² It might be a coincidence but the 2025 BachoTeX t-shirt was gray, which is an exception, because normally they come in rather primary colors.

³ Originally we had swapped green and yellow.

of a TeX run. Here we should note that by using a circle we add a clue: because we can color part of the circle, progress is seen not only in color but also by position on the circle.

We leave the ZigBee based HUE variant aside, except for mentioning that the device we will discuss now can also proxy to a ZigBee hub and that on my desk I have a portable Go that can be turned on in colors alongside the device. That gives the possibility for remote monitoring. One of the participants at BachoTeX mentioned that processing a 1000 page document took a very long time (and likely much more when the switch to LuaTeX had to be made) and although normally a ConTeXt user will not experience such runtime, one can imagine some visual indication that ‘The run has finished.’ or ‘The run failed.’ makes sense.

The idea is that we keep track of the state that ConTeXt is in when processing a document. We distinguish (at least) these scenarios:

- Most users only encounter a normal ConTeXt run. A newly created or heavily redacted document often takes a few runs to get all cross references, lists and various multi-pass data right. Small changes then need one or two runs. If we need more than nine runs we have a problem, and when the run fails we have an error.
- We ship a test suite that has thousands of files that also demonstrates some features. Processing that suite takes some time and for that reason the `mtx-testsuite` runner can run 8 jobs in parallel. At the end we get a report file mentioning the problematic files.
- Creating a distribution involves collecting files, running some scripts that generate manuals, zipping up files, creating installers, etc. From that perspective it is nice to get some feedback if a step fails.
- When processing multiple files the `--parallel` option can be of help. This is basically a collection of regular runs but some indication of progress and results is nice.

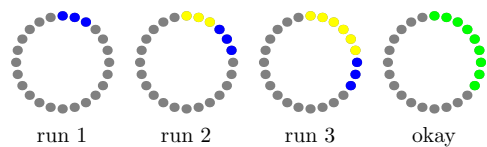
When it comes to problems, these can easily go unnoticed, for instance unknown references, missing files, absent images, overfull boxes. Future versions of this signals subsystem will deal with that too. An error is more dramatic because when a PDF file is written and an error occurs, we end up with an invalid file. For that reason ConTeXt produces an error document. But still, some additional feedback doesn’t hurt.

As it’s all about colors and signaling states, we need to mention what colors we use:

● busy	blue	the file is being processed
● done	yellow	processing (of an intermediate run) has finished
● finished	green	everything is fine after the last run
● problem	orange	something is wrong (like maxnofruns)
● error	red	we ended with an error

The `reset` state turns off all the lamps. When we need more than four runs the last lamp is reused. When we do a normal ConTeXt run we start out with lamp one being ●. When finished without issues that lamp turns ● and the next one goes ● and so on. When we have an error the current lamp goes red and normally ConTeXt has quit. When we need more than the maximum number of configured runs, nine by default, the last lamp is ●. When all are fine within this constraint we have ● lamps.⁴ There are ways to configure a different color palette for those who are color blind but that is still evolving.

We have one (squid), four (quadrant) or eight (segment) visualizers where each represents a run. As mentioned, by default ConTeXt limits the number of runs to nine so the last quadrant or segment is reused. In practice one never has that many runs, as it indicates some form of oscillation.

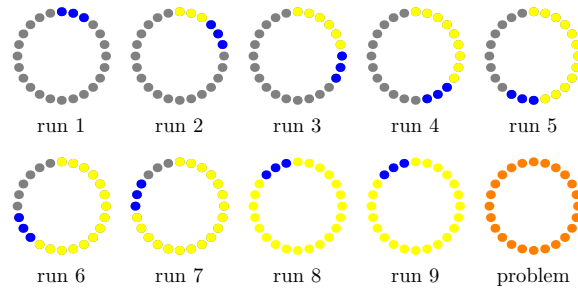


We use ● while we run and ● when a run ends. When we're done running we have ● when all went right, ● when there was an error and ● when we ran out of runs.



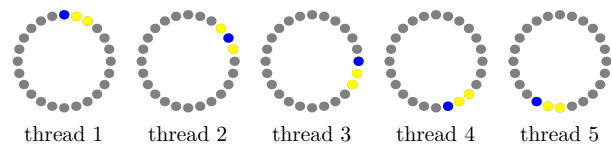
An error normally happens in the first run unless multi-pass data is involved but even then successive runs often fail in the first run (segment).

⁴ In the experimental setup ● and ● were swapped.

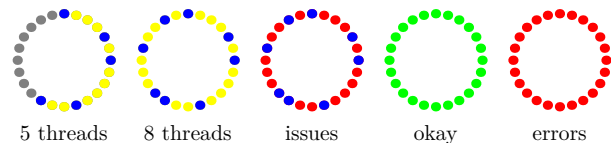


A problem normally shows after all segments are in use. However, in the future we might use the problem signal for more purposes.

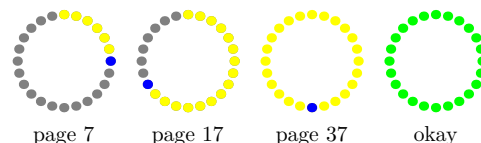
Segments are also used when we trace parallel runs. In that case each segment is bound to a process handler and within that the ● signal cycles over the segment. When there are many runs all segments show ● with active handlers showing a ●. When there is an error, the ● becomes ● and when all processes are finished we have either ● or ●.



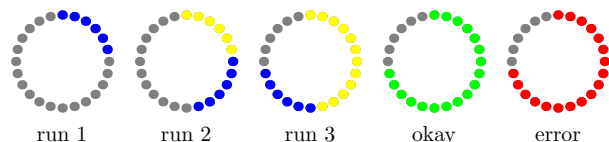
The segments shown above are combined into one as shown below. We use this feature when we process the test suite which has thousands of files but any `--parallel` driven run uses this feature.



We can also run in so-called squid mode. Here we have one segment spanning the whole circle where the ● is bound to the current page. We use this when we have documents with hundreds or even thousands of pages. In that case the signal will run around the circle. You might see a pause or slowdown depending on what happens on a page.



Quadrants are larger segments and often nicer when the number of runs is limited. Again we use ● and ● to show progress while the final result is reported with ● when we succeed, ● when we need more runs or ● when something went wrong.



Quadrants differ from segments in that they show more details about the current run: within a segment we trace pages as we do with squids. Of course the granularity is limited because a squid has 24 LEDs and a quadrant only 6. Anyway, detail comes at the price of a little more runtime.

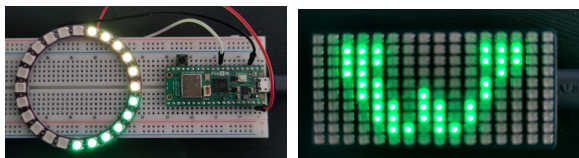


Figure 5: Left: the prototype; right: a variant.

5 Variants

We are also playing a bit with variants in feedback and because this gadget will be part of a workshop at the ConTeXt meeting we also expect some discussions. For instance there is a variant that can also display a character, most noticeably using Don Knuth's bitmap 36 font. That means that we can actually consider feedback with a letter or even emoticons about what is wrong, like missing stuff (think graphics, fonts, files, characters) or dubious cases (like overfull boxes). Because this font is proportional we need a 8 times 12 LED setup.

6 The PDF trickery

This is a good moment to come back to where it all started: Tove's presentation. So how did we synchronize colors with flag colors? We're talking PDF and although in the past one could do nice tricks with presentations, over time less has become possible. There is no usable communication channel that doesn't spoil the document view. Also, even if we could hack something into Acrobat viewing, it's not portable to the viewers that we use (like Sumatra and Okular).

The solution we came up with is the following. We have a script that runs in the background. That script queries the task manager (using some shell command) for the current title of the viewer. The filename of the current document results in a lookup in a (Lua) table that tells what colors relate. So, the 24 flags in the presentation were in 24 two-page documents (a flag and a hello page) and hyperlinked to a sequence. So, a change in file triggered a change in the four color bulbs because the script sent HTTP

requests to the ZigBee (HUE) hub. Because we have an interfacing library that script is currently talking to the device that thereby provides a bit more abstraction than sending requests to a ZigBee controller. In the end surprisingly little code is needed.

We didn't mention yet that we don't need special programs. In LuaMetaTeX we can send HTTP requests, either using the built-in socket library or by using curl (as program or library). The latter is needed when we need secure connections to a HUE hub. The device however is serial. Again we don't need special programs because LuaMetaTeX provides a simple `serial.write` helper.

The previously discussed signaling modes are orchestrated by the ConTeXt runner script that for each run (as there can be multiple) sending state commands (reset, busy, done, finished, problem, error), either or not for parallel runs. However, during the run ConTeXt itself can also send states, like progress by page. There is of course a little extra overhead but because ConTeXt runs are quite fast, little is lost.

7 Configuration

The ConTeXt distribution comes with a manual that describes the device, explains the (simple) serial commands that do the job, tells how to configure the lot, and has examples of a few applications, like squid, quadrant, segment, and parallel tracing. The test suite and distribution generators also are signal aware. There are of course some scripts that can be played with (or called from other applications to signal states).

To give a taste of a setup, this goes into a file `ctxsignals.lua`, which has to be in your local tree. Here we also define a HUE setup.

```
return {
  comment = "signal setup file",
  servers = {
    squid = {
      protocol = "serial",
      port     = "COM7",
      baud     = 115200,
    },
    hue = {
      protocol = "hue",
      token    = "....."
      .. ".....",
      url      = ".....",
      lamps   = { 3 }
    },
  },
  signals = {
    squid = { enabled = true,
              lamps   = { 4 } },
```

```

    quadrant = { enabled = true,
                  lamps = { 3, 4 } },
    segment  = { enabled = true,
                  lamps = { 3, 4 } },
  },
  usage = {
    enabled = true,
    server  = "squid",
  },
  version = 0x1.004189374bc6ap+0,
}

```

Out of the box we don't enable the HUE connection, so you need to set that up, for instance with

```
mtxrun --script squid --configure --hue
```

The state can be saved in the device if needed. You then also need to configure WiFi:

```
mtxrun --script squid --configure \
  --wifi --ssid=... --psk=... --connect
```

A test can be done with :

```
mtxrun --script squid --test --squid
```

Normally you will program this in Lua but direct control is also possible:

```
cmd="mtxrun --script squid --signal --segment"
$cmd --reset      --verbose
$cmd --busy       1 --verbose
$cmd --done       1 --verbose
$cmd --busy       2 --verbose
$cmd --finished  2 --verbose
```

The sent sequences are `sr0`, `sb1`, `sd1`, `sb2` and `sf2`. When you also track pages the device itself keeps track of what led in a quadrant has to change in which case we show a stepwise progress instead of a static busy strip. There are various examples in the distribution that show how to program signals.

8 Wrapup

In case you wonder how bound this is to ConT_EXt ... there was a time (when I started attending T_EX meetings around 1995, thirty years ago) when the fact that ConT_EXt had some scripts for managing runs, formats, fonts, etc., was considered impractical and strange. But times have changed and (maybe thanks to Lua) distributions now install plenty of scripts. Who knows where this trickery will show up a few decades from now.

As a bonus, here are a couple of related pictures from the 2025 ConT_EXt meeting:



Mojca Miklavc designed a PCB (with esp32 that has serial and wifi), and constructed the pieces using a 3D printer. The custom wooden stand was made by Willi Egger. At the meeting, we started with an assembling workshop to put the pieces together and then flash the software; the serial connection worked ok on Windows, GNU/Linux, OSX and FreeBSD. We now have interfaces for sending commands and setting the LEDs. Once everything is stable all the resources will be in the distribution (schematics, 3D recipe, code, etc.).

So, where we started with mentioning tagging related to PDF, we ended up with tagging T_EX runs instead, which is way more fun! If it wasn't for Tove's suggestion and research we'd not have investigated this new ConT_EXt feature. As a bonus for BachoT_EX attendees, she helped bring down the average age to below retirement age.

Playing with this and programming the device at least made me feel a bit younger because it reminded me of the early times of messing around with micro-processors. However, it must be said: these tiny machines we have nowadays are way more powerful than their predecessors. Just like today's demanding T_EX engines and jobs make one wonder how Don Knuth managed to get it rolling, these devices make one realize how far we've come in combining technologies. One wonders how the PDP-11 machines of those times compare to the small Pico 2 of today.

◇ Hans Hagen
Pragma ADE