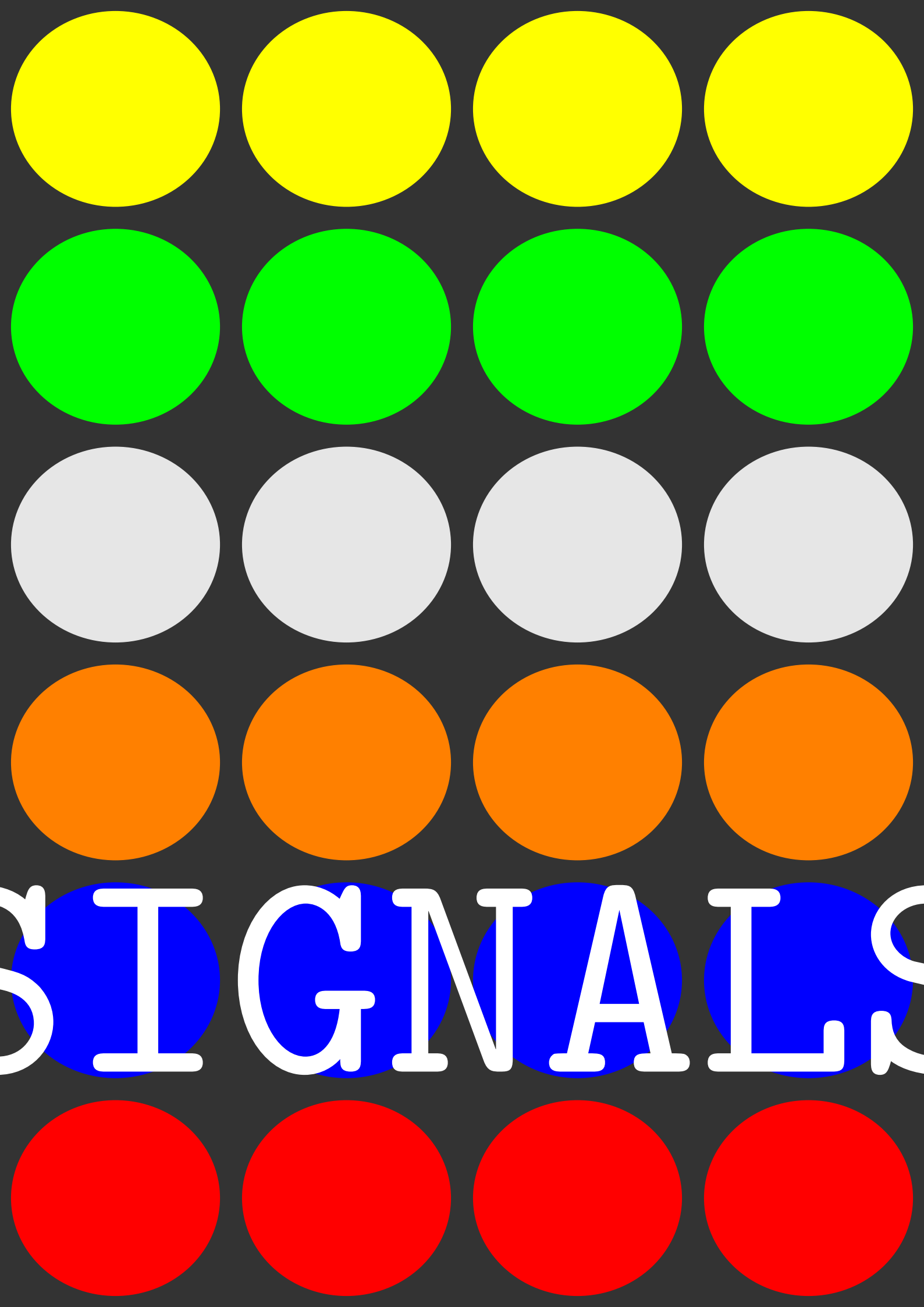

SIGNALS

1 The rationale

On the average a TEX run is pretty fast. On a 2018 laptop processing the 650 page LuaMetaTEX manual takes some 15 seconds per run and a LuaMetaFun manual demands a similar amount of time once the ridiculous large contour graphics are cached and that manual is not representative anyway. Mikael can process a 300 plus page math book in under 10 seconds on his equally old chromebook, and his shiny new 2025 one needs some 7 seconds. When my colleague processes colorful educational math books from (many small) xml files and fancy makeup with images all over the place, it makes little sense to get a coffee while waiting for the run to finish because even complex runs seldom need more than a minute; it depends on the number of runs needed.

Processing the more than 2000 files in the test suite takes 750 seconds with the parallel runner option enabled. A hardware upgrade (or using a desktop machine) might speed that up but even then I'm not going to stare at the screen for minutes. Seeing a summary is good enough. So there one has to wait in order to identify potential issues. Normally that run happens in the background which brings the danger of forgetting to check the final error report. Ideally that suite should runs without issues before we update.

Often, running a large job in the background can make sense so how then to keep track of potential issues and progress? While preparing a talk for BachoTEX 2025 we played with color lamps as part of a presentation where colors and turning lights on and off was determined by file on display: flagged pdf. That made us wonder if we can also use such visual clues for the run state and therefore ConTEXt signals saw the light!

At BachoTEX it was decided to follow up on this at the ConTEXt meeting, this time with a more practical approach than a ZigBee setup: a dedicated device connected to the usb port. Actually, in the end that one also was made to proxy to a ZigBee hub so we got the best of two worlds.

In the following sections we describe the various possibilities: the original approach of using ZigBee devices, and the more practical follow up using a small device, nicknamed Squid.

2 The states

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. In 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 get unnoticed, for instance unknown references, missing files, absent images, overfull boxes. Future versions of signals will deal with that too. An error is more dramatic because when a pdf file is written and an error occurs, we en dup with an invalid file. For that reason ConTEXt produces an error document. But still, some additional feedback doesn't hurt.

The ZigBee approach, the one we started with, introduces a color feedback palette but the same colors are used in the Squid as well. Tracking the scenarios mentioned here are implemented using the same palette:

●	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 lamps 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.[1] There are ways to configure a different color palette for those who are color blind but that is still evolving.

[1] In the experimental setup ● and ● were swapped.

3 A ZigBee setup

It all started with a ZigBee based setup, a low power mesh network with devices like lamps, switches, sensors and whatever. For signaling the state of a run I used four hue color lamps that came into my possession but for which I never had a purpose. However, any amount of lamps will do. Actually, I ended up with a single portable lamp so that I can keep track of the state of a run away from the machine it runs on.

> For the record, already for ages I use the warm white hue bulbs combined with motion detectors and switches in the office rooms and living spaces. For that we employ two hubs for communication but for performance reasons we always ran our own controller script, written in Lua using LuaTEX as Lua engine as it has socket support and can communicate http. That way we got around the limitations of that time: the hubs have a low performance and couldn't handle multiple sensors per room. In fact we could handle more devices than officially supported.[2]

Announced changes in the hue api and accessibility of those hubs plus the fact that one has to be online when initializing a hub made me look at some fallback solution, just in case. Already for a while, I had a few Phoscom ConBee sticks in stock and I decided hat this was a good moment to play with them. Such a usb stick plugs into the laptop that I use and thereby is independent on the hue hub which actually improves performance: a state switch takes some 25 milliseconds. Of course there is still some management software idling in the background which is sub-optimal. So, to summarize this last setup, we have a usb communication hub, plugged into a computer that runs the related management software, which comes with a web interface for registering devices. We connect (in our case) four lights and in order to be able to turn lights off independently I registered two buttons.

> Although the Philips hue lamps don't come cheap, they are of good quality. After testing with the E27 lamps we decided to get a few (well built) Go Portable Lights and these also work well as state lights. The build quality is better than similar portable ZigBee devices.

In the end a regular hue setup was more practical. For testing I used an old FritzBox WiFi router to which I connected a hue hub. A disadvantage is that one needs to be on that network so eventually it's best to be on the usual WiFi router and use a hue hub on that instead. Later we will see how the Squid can avoid this dilemma.

> We tried to use cheap Ikea switches but they tend to disappear or drain batteries when a ConBee stick is not active. It should be possible to use their devices but the hue hardware bulbs and switches are of good quality so why bother. In the experimental setup the larger bulbs were hidden in opal white glass balls and were sitting next to the 65 inch monitor that I used (@ 3 meter distance); one quickly gets accustomed to this!

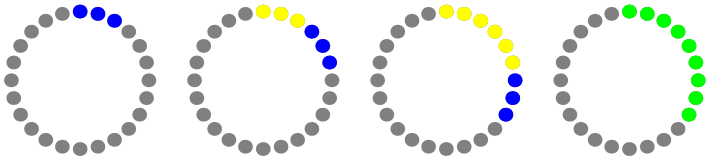
[2] In a similar fashion we abuse LuaTEX plus script to control a 12 zone EvoHome heating setup, again to get around limitations. So, here the long term stability of LuaTEX combined with a relative rich set of libraries that come with ConTEXt pays off.

After using the the test setup at BachoTEX with a dedicated wireless network and ZigBee hub(s) we decided to follow up with a dedicated device connected to the usb port. This was in the end more reasonable than using WiFi because client can be isolated from each other and thereby a WiFi controlled approach is fragile. The device actually can be set up to respond to WiFi signals but that is mostly an experiment. Less experimental is the ability to act as proxy to a hue hub in parallel to running in circles (Squid), which we will discuss in the next section. The main hurdles in a serial connection are figuring out what port to configure and making sure (on linux) that the user can access the port.

4 The Squid

We have now arrived at the alternative: a usb connected device. We have a set of addressable leds, for instance a circle with 24 so called WS2812B leds or a rectangle with 10 times 16 leds (or more likely: 8 times 12). These are what we call device 1 and device 2 Squid. The reference is device 1, the one that was introduced at the ConTEXt 2025 meeting. The device is driven by a RPI Pico for which the distribution has software so you can assemble your own (the first tests used a large silicon led strip).

Assuming a device 1, we have one (squid), four (quadrant) or eight (segment) visualizers where each represents a run. 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.

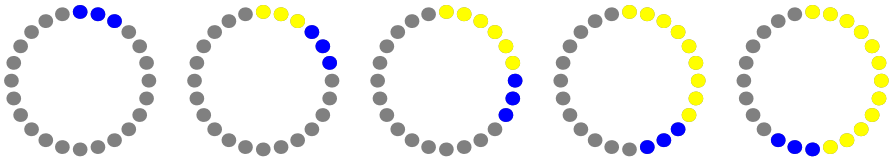


run 1 run 2 run 3 okay

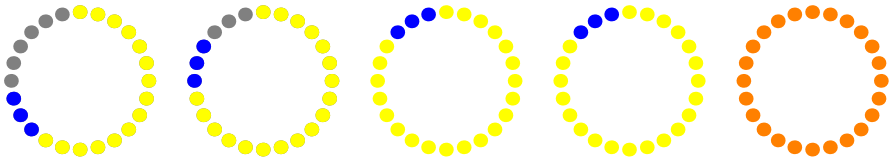
We use ● when we run and ● when a run ends. When we're done running we have ● when all went right, ● when there was an error and ● when we ran out of runs.

error

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



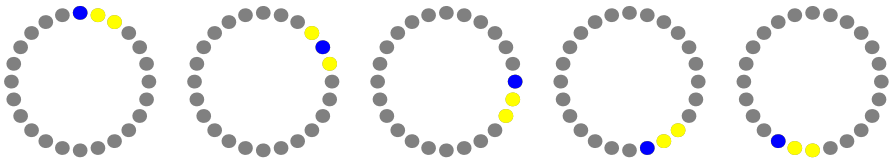
run 1 run 2 run 3 run 4 run 5



run 6 run 7 run 8 run 9 problem

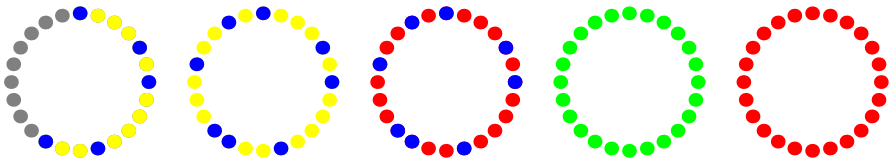
A problem normally shows after all segments are in use. However, in the future we might use the problem signal for more purposes.

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 either have ● or ●.



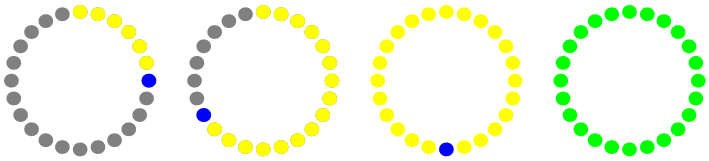
thread 1 thread 2 thread 3 thread 4 thread 5

The segments shows 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 use this feature.



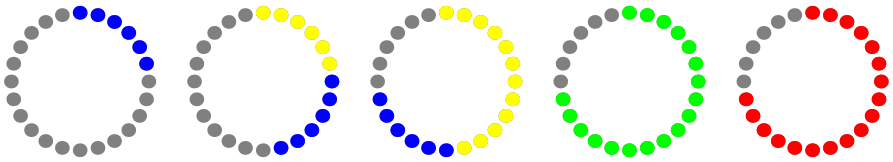
5 threads 8 threads issues okay errors

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.



page 7 page 17 page 37 okay

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.



run 1 run 2 run 3 okay error

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.

For the curious: a device 2 uses a rectangular setup and therefore can also show (single) characters in Knuths 36 font. This is more of a a playground.

5 The software

So how is this controlled? Triggering signal events is done with either the `context` script or with `mtxrun`:

```
context --signal=runner ...
mtxrun --script signal ...
```

Of course the `runner` signal has to be enabled and later we will show how that is done. You can also add a directive at the top of your document:

```
% signal=runner
```

When you use a Squid the communication with the device goes via the serial (usb) interface using simple command sequences to limit the overhead. We started out with the quadrants and later added segments (for more granularity) and squids (single but with page details) and in the end the (for the user) hidden interfaces became rather similar. In this case you can use these directives:

```
% signal=squid
% signal=segment
% signal=quadrant
```

When you use a Squid the `runner` directive is equivalent to the `segment` one. What actually happens also depends on the configuration file. Just in case one wonders about lights, sound is another option as are messages to ones phone but we avoid being too fancy.

At BachoTEX we introduced a compromise: QR code. There is no status information then but an error will create a pdf file with a QR code:

```
% signal=qr
```

The configuration file can look a bit intimidating but if you only use the Squid some entries can be omitted. Here is an example of such a file (it kind of evolves so it might change a bit at some point):

```
return {
    servers = {
        squid = {
            protocol = "serial",
            port     = "COM6",
            baud     = 115200,
        },
        -- optional, not needed when device is used
        conbee = {
            protocol = "deconz",
            token    = "XXXXXXXXXX",
            url      = "http://127.0.0.1",
        },
        -- optional, not needed when device is used
        hue = {
            protocol = "hue",
            token    = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
            url      = "https://192.168.178.25",
        },
    },
    signals = {
        runner = {
            enabled = true,
            lamps   = { 1 },
        },
```

```
        squid = {
            enabled = true,
            lamps   = { 1 },
        },
        quadrant = {
            enabled = true,
            lamps   = { 1, 2 },
        },
        segment = {
            enabled = true,
            lamps   = { 1, 2, 3, 4 },
        },
    },
    usage = {
        enabled = true,
        server = "squid",
    },
    version = 1.001,
    comment = "signal setup file",
}
```

This file has to be someplace where it can be found by ConTEXt and its scripts, for instance in `TEXMF-LOCAL/tex/context/user/mkxl`. You can make an empty one with:

```
mtxrun --script signal --create
```

Successive runs will check for the presence of that file and use it when found. Otherwise the current directory is used to save the new configuration, so you'd best move it to where it belongs.

Servers are configured with (for instance):

```
mtxrun --script signal --servers \
    --protocol=serial --port=... squid
mtxrun --script signal --servers \
    --protocol=hue --url=... --token=... hue
mtxrun --script signal --servers \
    --protocol=deconz --url=... --token=... conbee
```

and enabled with:

```
mtxrun --script signal --usage squid
```

The various signals are set up with:

```
mtxrun --script signal --signals --lamps=1,2,3 runner
mtxrun --script signal --signals --lamps=1     runner
mtxrun --script signal --signals --lamps=-4    runner
```

Positive values indicate (color) lights, negative values indicate plugs that only support on/off states. A specific signal has to be enabled:

```
mtxrun --script signal --enable runner
```

6 The prerequisites

This section is only relevant iF you use ZigBee without a Squid. The Squid can be configured to also control hue lamps for which it has to be connected to the relevant WiFi network.

The signal ZigBee handler assumes that curl is installed, the binary and/or library. We also need a ZigBee gateway, like hue or ConBee (that last one needs the Phoscom app deconz has to be run). In the case of a hue system the hub should be on the same network. You need to configure lights and then register these as shown above. In order to access the ZigBee interfaces you need to get a token. For instance in deconz:

```
curl -X POST -i "http://127.0.0.1/api" \
    --data "{ \"devicetype\" : \"password\" }"
```

For hue you need something like this (first hit the button on the hub):

```
curl -X POST -k -i https://192.168.178.25/debug/clip.html \
    --data "{\"devicetype\":\"service-name#username\"}"
```

In the future I might add some more protocols, for instance FritzBox also supports ZigBee now but I can't test that right now.

7 The interface

There are a few interfaces that one can use to construct dedicated state reporting, as we do for the test suite and distribution. For running ConTEXt there are also various predefined setups. So, normally a user can just rely on what we provide out of the box. It is however good to realize that there are two fundamentally different usage patterns:

- The runner and segment signals are controlled by the `context` script. This means that the overhead is minimal and that the run itself is not aware of it being signaled.

- The squid and quadrant signals show progress per page and for that the run itself needs to provide information. Here the `context` script initializes the Squid and the ConTEXt run then updates the state. This brings a bit of extra overhead but it's probably worth it.

So, when you process a file in the usual way, the runner will act upon the return state of a TEX run. Because the runner knows how often it has ran and when it's finished, either or not successful, it can manage the initial and final signals best and also knows when to proceed to a next segment or quadrant. Below we show some ways to control signaling but keep in mind that normally all happens automatically in the most efficient way possible.

Here we just give a simple example so that you get an idea what happens behind the screens. The slow way is to call the runner, like:

```
os.execute("mtxrun --script signal --state=reset runner")
```

But you can also use this pattern, assuming that you run ConTEXt or use a Lua file with `mtxrun --script somename.lua`.

```
local signal   = utilities.signals.initialize("runner")
local enabled  = type(signal) == "function"

if enabled then
    signal("reset")
end

-- code that sets 'okay'

if enabled then
    if okay then
        signal("done",1)
    else
        signal("error",1)
    end
end
```

The second argument is the lamp number (or run) and the third argument indicates that we apply the state to all lamps. The `run` and `finished` states always apply to all lamps. In the `util-sig-imp-*` files you can see how the various signals are set up. The differences are subtle. It only makes sense to study this when you have workflows with special demands. In principle you can create a setup with multiple squids and create visual feedback as you wish.

8 The Curl library

If you use the Squid you can forget about this section but otherwise we assume that the Curl library `libcurl.so` or `libcurl.dll` is present, preferable in the TEX tree under `bin/luametatex/lib/curl`. If that one can't be loaded the program will be used instead. We could fall back on the built-in socket library but we have to assume that servers use `https`.

9 The runner

We have two scripts, the original runner related script (signal) and one for the device (squid, quadrant and segment). The runner script can be used to control lights. Here are a few examples. This will turn off the lights:

```
mtxrun --script signal --reset
```

Here you get some information about the current setup:

```
mtxrun --script signal --info
```

A state can be set with:

```
mtxrun --script signal --state=busy         runner
mtxrun --script signal --state=busy --run=2 runner
mtxrun --script signal --state=busy --all   runner
```

Help information is available with:

```
mtxrun --script signal
```

Remark: the `--prune` and `--wipe` options are only for testing as they mess with rules. Turning off a lamp can be done with `--reset` anyway.

10 The Squid hardware

A Squid is basically a small controller like a RPI Pico connected to a series of leds. The software is written in the Arduino editor and uses related libraries. You can just compile it for any similar device in which case you need to configure the right io pins. It's a relatively simple setup. With (2025) led rings and controllers being cheap you can in principle make a Squid for less than 15 euro but you might want to spice it a bit with a switch. We run the leds at low brightness so there is no need for additional power.

(todo: reference to design document and some photos)

11 The serial interface

The serial interface uses simple command sequences: the first character tells what we want to do, the second character is a specifier within that category. Follow up characters are data, like a specific action (busy, done, etc.) optionally followed by a number (segment, quadrant, etc.), or some configuration number or string. In all cases ending with a return `\n` avoids getting out of sync due to look ahead confusion.

We can set all leds with the `a` command. The other commands can do the same but here a step is equivalent to busy.

`a [i r]`	reset (also initializes counter)
`a s`	step (busy)
`a [d f p e r]`	set state

Segments are controlled with:

`s [i r] [0 1..8]`	reset (also initializes counter)
`s s [0 1..8]`	step (increments)
`s [d f p e r] [0 1..8]`	set state

Quadrants are controlled with:

`k [i r] [0 1..4]`	reset (also initializes counter)
`k s [0 1..4]`	step (increments)
`k [d f p e r] [0 1..4]`	set state

Single led squids are controlled with:

`q [i r]`	reset (also initializes counter)
`q s`	step (increments)
`q [d f p e r]`	set state

When configured, hue bulbs can be turned on and off by:

`h [d f p e r] [0 1..4]`	set hue color light

It's possible to reset the lot:

`r`	reset

We don't go into detail about this but there's also direct control of leds:

`d <n> r g b`	set led directly by four byte values

Here is a variant of this (no comment as it's experimental too):

`i <n> r g b`	set individual led by four byte values

The device is set up with the next set of commands. The configuration can be saved on the device itself. At some point we will add an option to configure using the configuration file.

`c s`	save
`c f`	format
`c w r`	wifi reset
`c w c`	wifi connect
`c w i str`	wifi ssid
`c w k str`	wifi psk
`c h r`	hue reset
`c h t str`	hue token
`c h h str`	hue hub
`c h l num`	hue lights (upto 4)
`c c [0 1 2]`	color
`c m s`	mode serial
`c m h`	mode serial + hue
`c m w`	mode serial + wifi
`c m b`	mode serial + bluetooth

If you have a device 2, there's also a text option, where the character set is limited to the repertoire in Knuths 36 font. This is more for myself to play with. You can also send text (string) via the signal script.

`t [d f p e r] character`	show this character

12 Connecting

On MS Windows you can plug in a serial (usb) cable and the device gets a port assigned. Normally you get the same port. You can check the right name (something `COM:`) in the device manager, so for example `COM6:`.

In linux a RPI Pico will normally be `/dev/ttyACM0` but there we also need to make sure that we get access. You need to be a member of the `dialout` group, which can be done with `sudo usermod -a -G dialout yourname` (you need to re-login).

When you connect the device visa for instance a usb hub in a keyboard you should expect uncomfortable delays, at least that is what I encountered on the linux box.

The software on the device will be part of the ConTEXt distribution and updates are to be expected.