

From Pure Code to Physical Controls

Lucas Sifoni / Code BEAM Europe 2026

Bureau Brut | Fonts | by style | by family | def fonts(), do: :print

Boogy Brut Collection — Buy

Boogy Brut — Buy

def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print

Boogy Brut Poster — Buy

def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print

Boogy Brut Wild — Buy

Droulers Collection — Buy

Droulers — Buy

def fonts(), do: :print
def fonts(), do: :print

Droulers Clarendon — Buy

def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print

Matorral — Buy

def fonts(), do: :print
def fonts(), do: :print

Round Collection — Buy

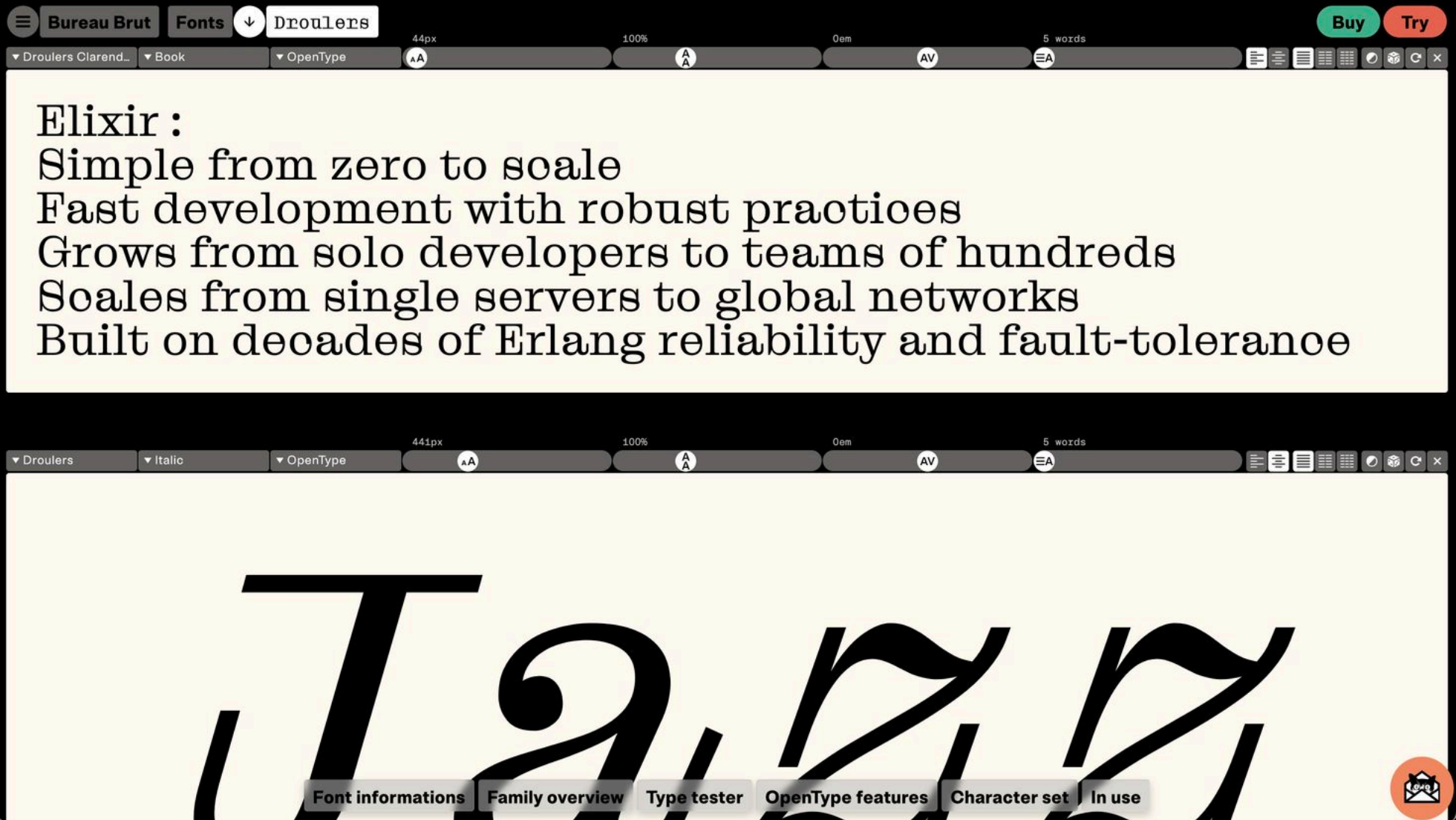
Round — Buy

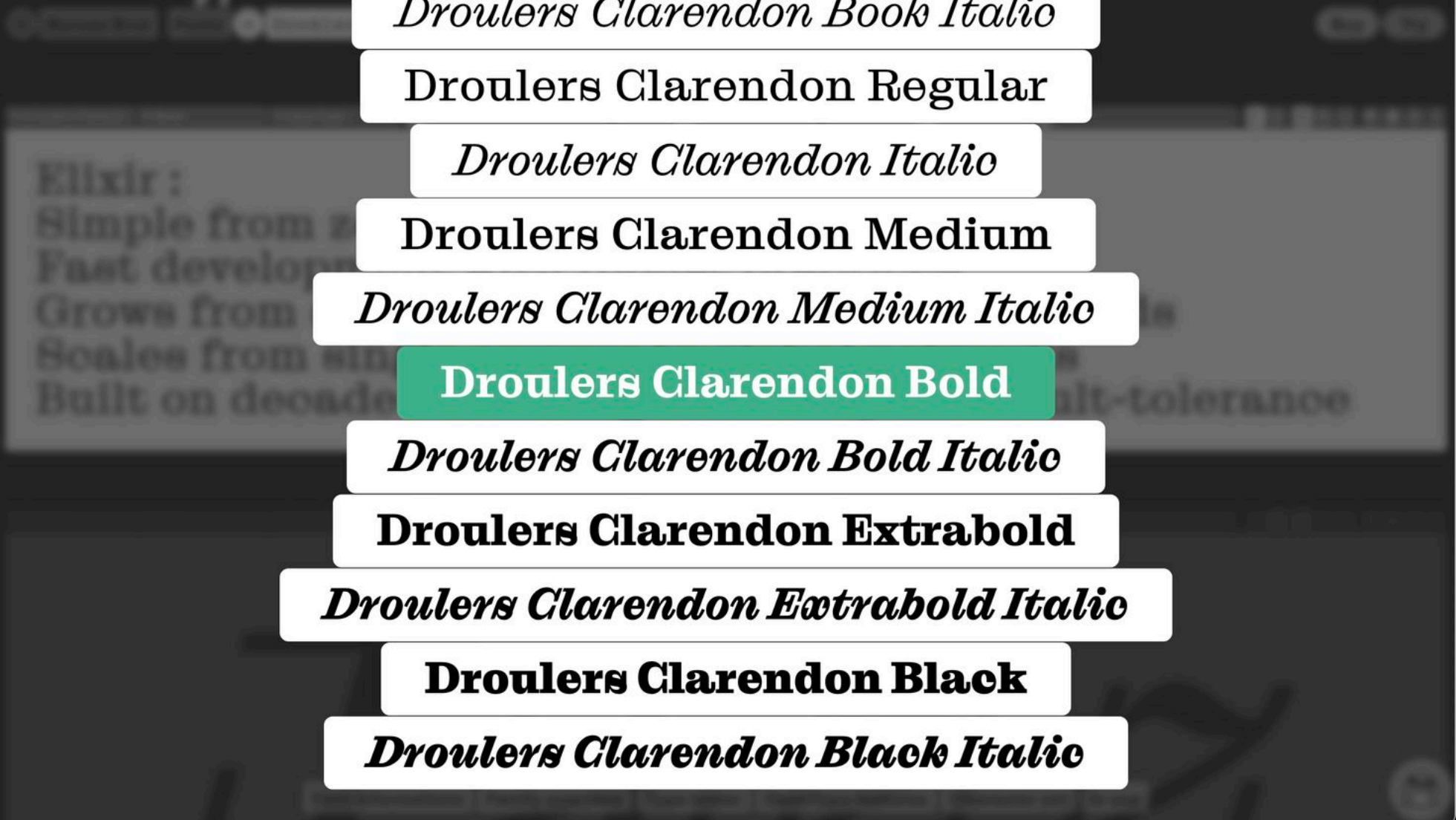
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print

Round Condensed — Buy

def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print
def fonts(), do: :print

Elixir :
Simple from zero to scale
Fast development with robust practices
Grows from solo developers to teams of hundreds
Scales from single servers to global networks
Built on decades of Erlang reliability and fault-tolerance




Droulers Clarendon Book Italic
Droulers Clarendon Regular
Droulers Clarendon Italic
Droulers Clarendon Medium
Droulers Clarendon Medium Italic
Droulers Clarendon Bold
Droulers Clarendon Bold Italic
Droulers Clarendon Extrabold
Droulers Clarendon Extrabold Italic
Droulers Clarendon Black
Droulers Clarendon Black Italic



Craft by the people
and for the people ;
as a happiness to the
maker and to the user
William Morris, 1879


Brutator
Aa


:glyphs
Aa
Aa
Aa
Aa
Aa
def fonts(), do:
:print

Project guidelines

- Intuitive & playful to use
- Build instead of subcontract
- Design controls intuitively
- Build along the design
- Old school look and feel, powered by Elixir

Used by ~500 people to create ~2000 printouts in 4 days

Final hardware

- 196-key keyboard (self-built)
- Arcade trackball (adapted to run on 3v3)
- 3 joysticks & analog needle gauges
- A backlit keypad (self-built)
- Two switchboards, 30 switches with 60 indicators (self-built)
- A fake floppy disk reader (NFC tag reader)
- A Bixolon XD5-43d sticker printer

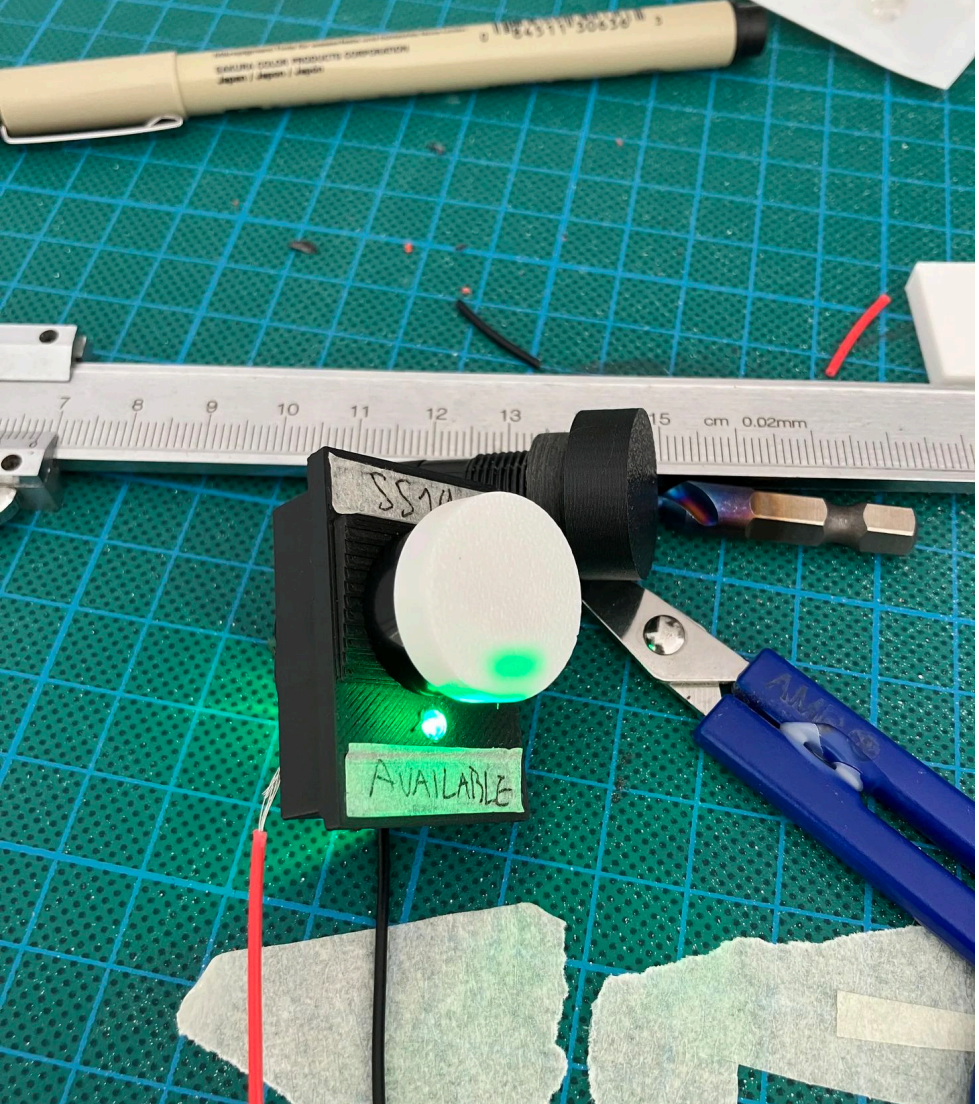
Trivia :

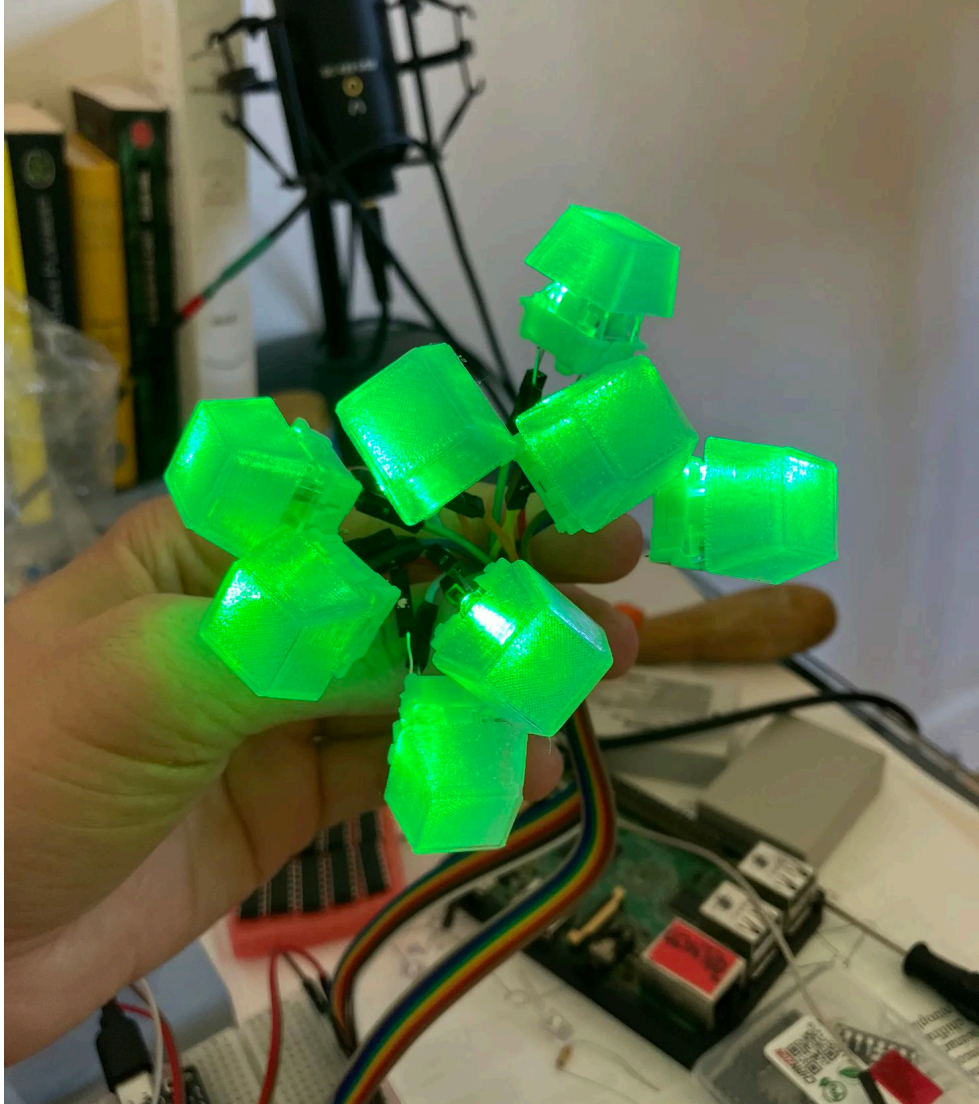
- Shift registers chains instead of IO expanders
- DIP ICs and passive components
- Hand wired on padboards to accomodate the short deadline

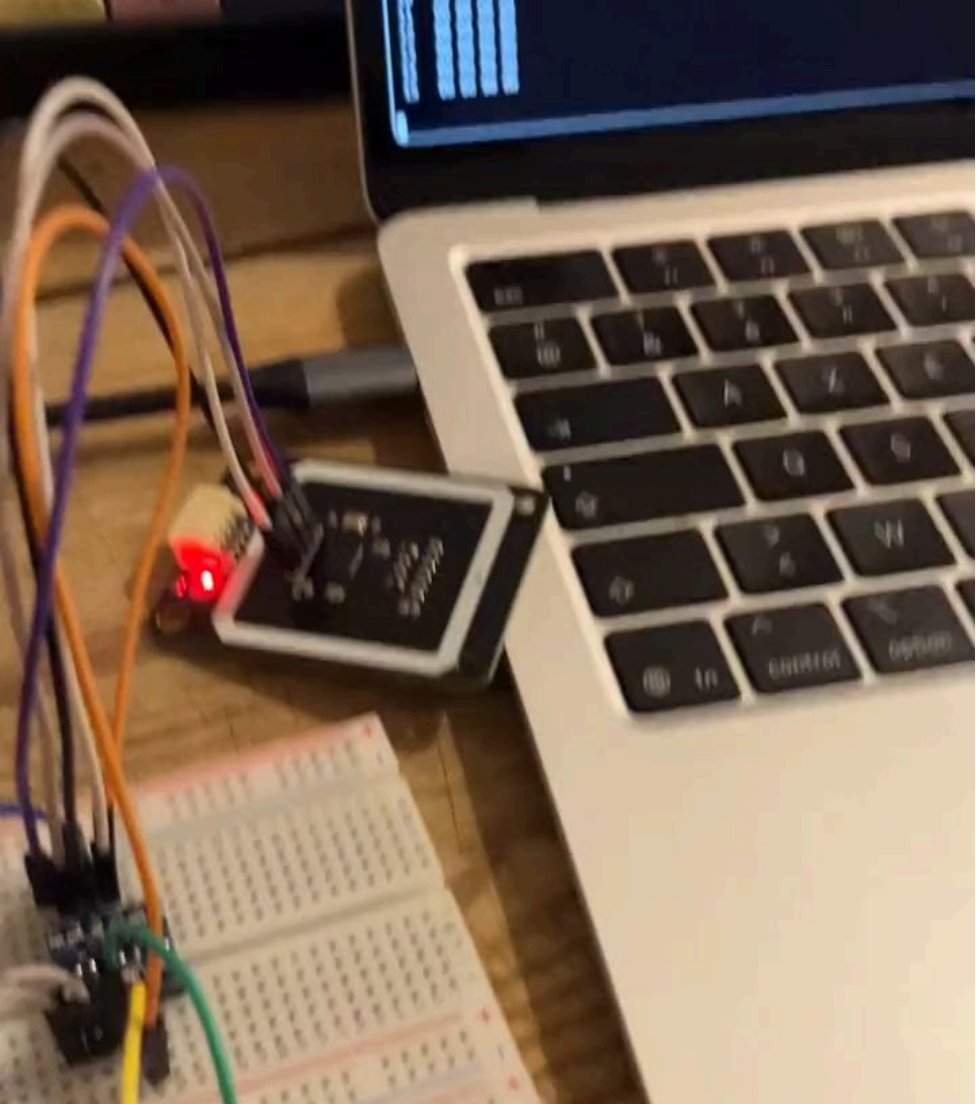
4 fully independent modules, 1 ESP32 each

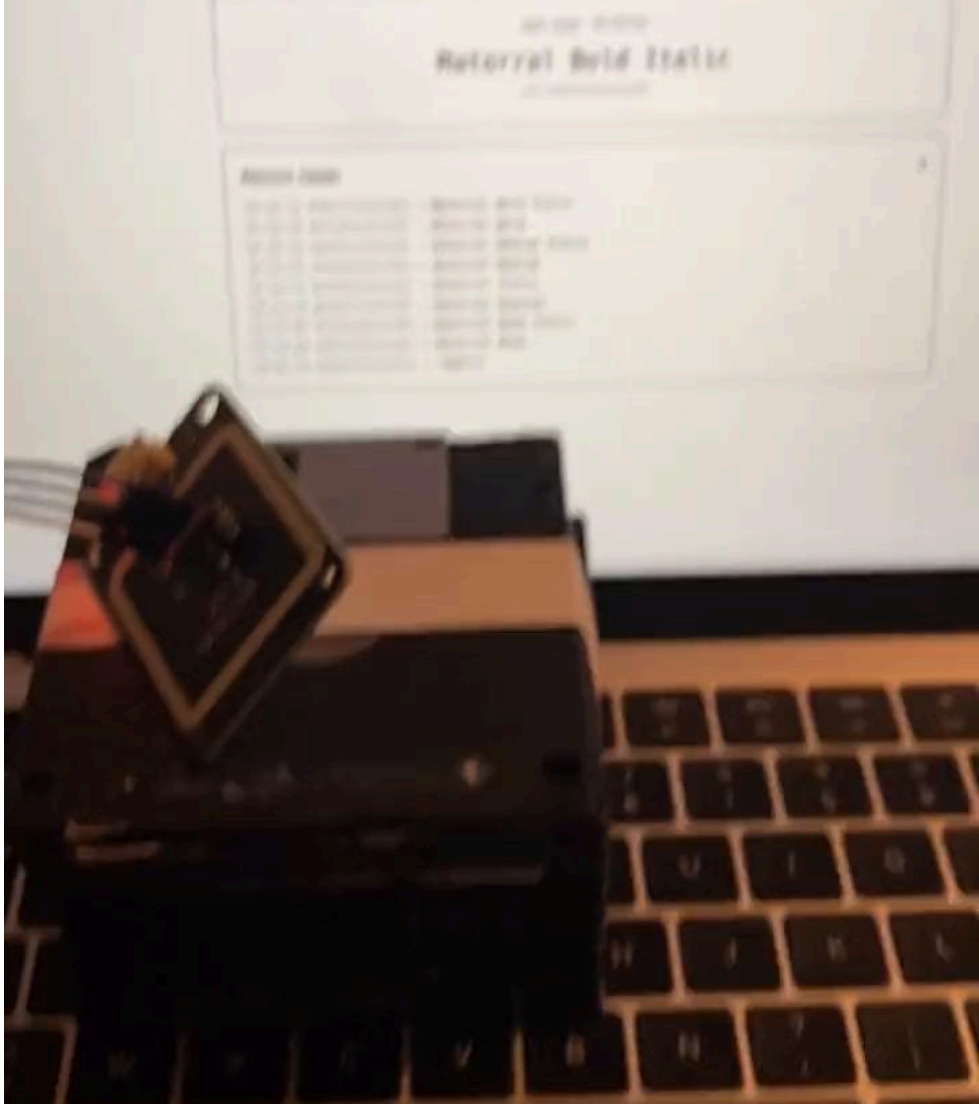


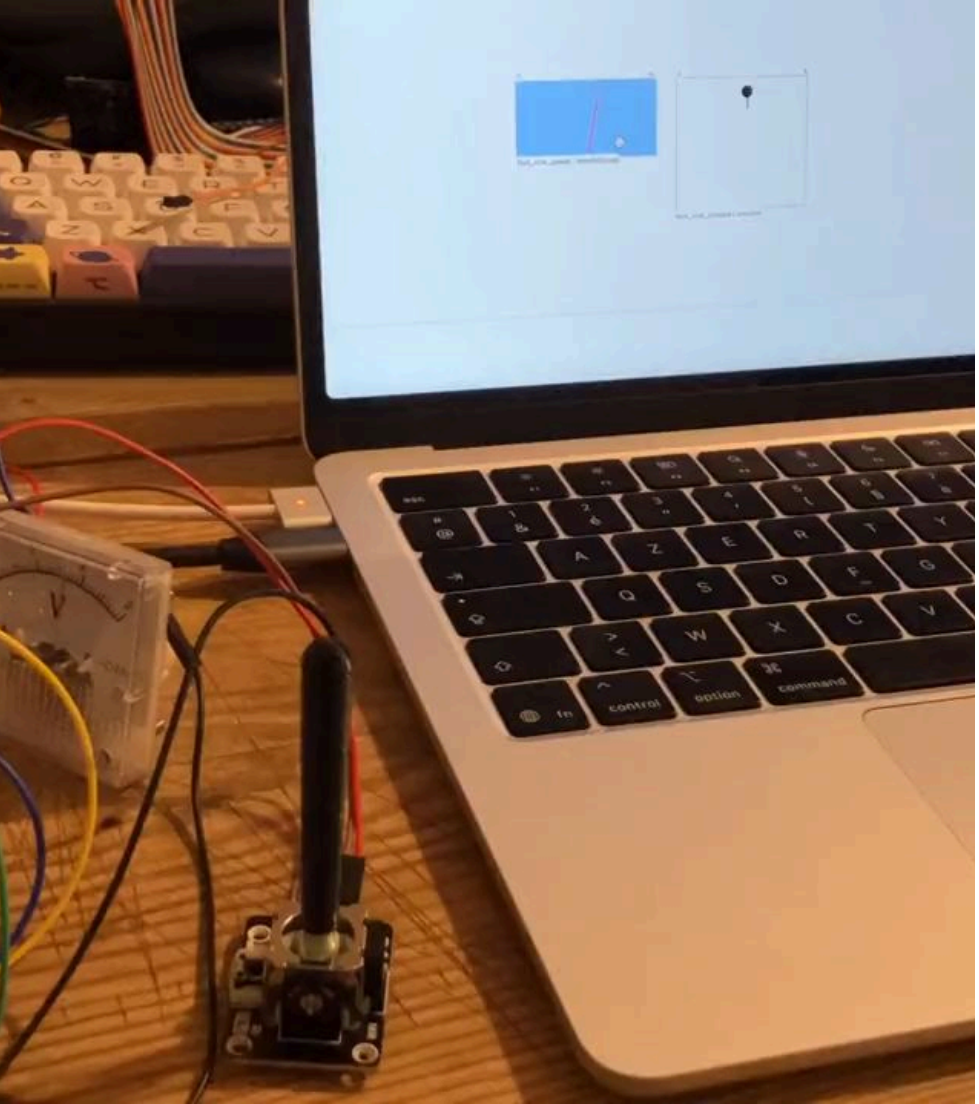

Integrated Circuit
P/N:74HC595N
PKG CASE:DIP-16
QTY:10
CHANZON
W57-1=10
10PCS ULN2803APG ULN2803A ULN2803 DI
P18 100% NEW
16


AVAILABLE



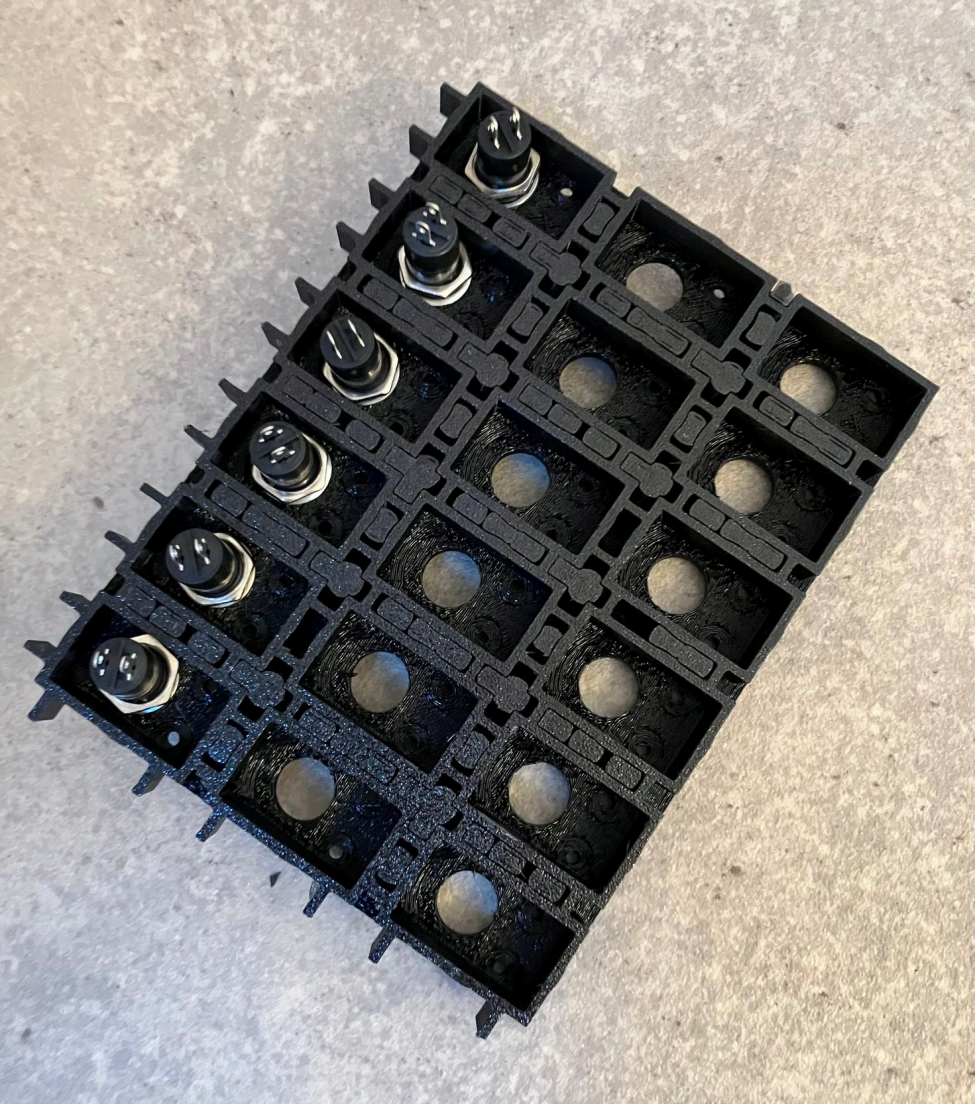


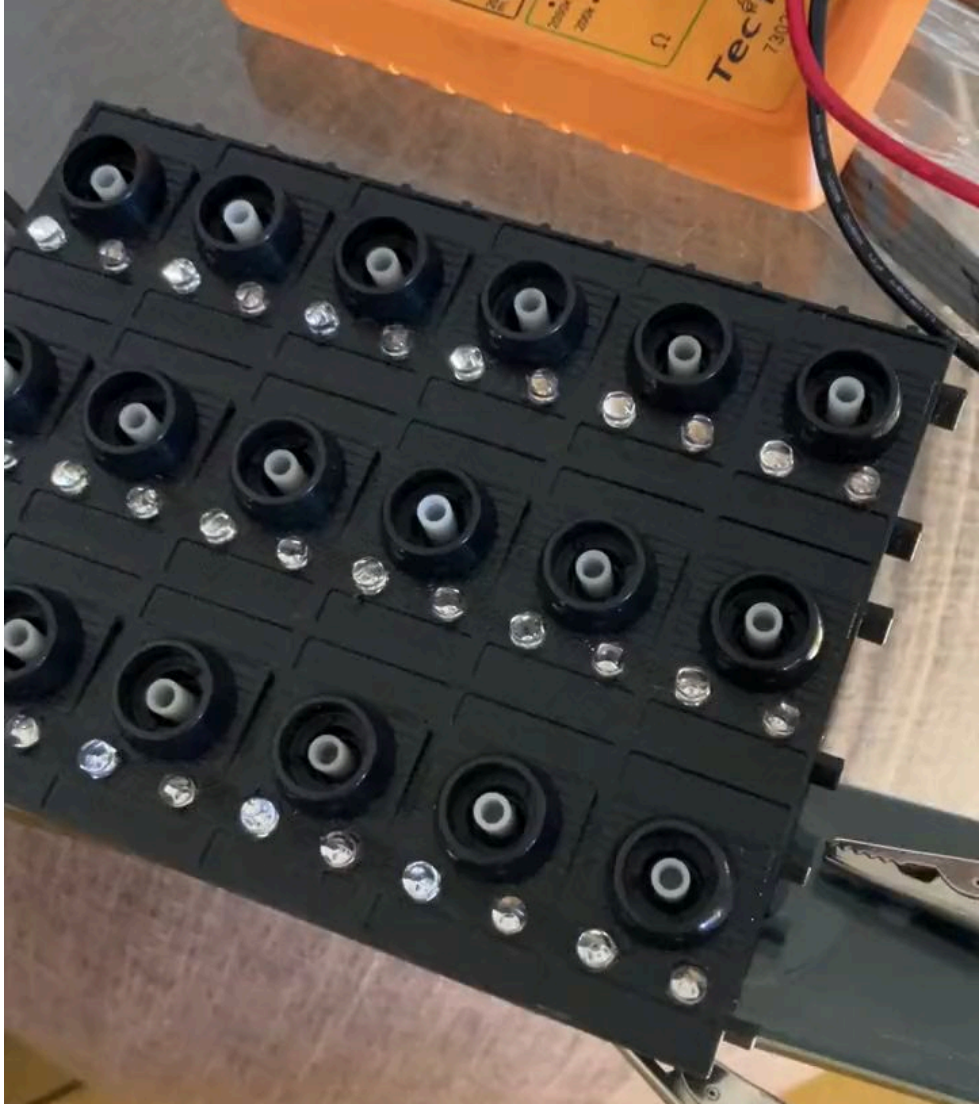


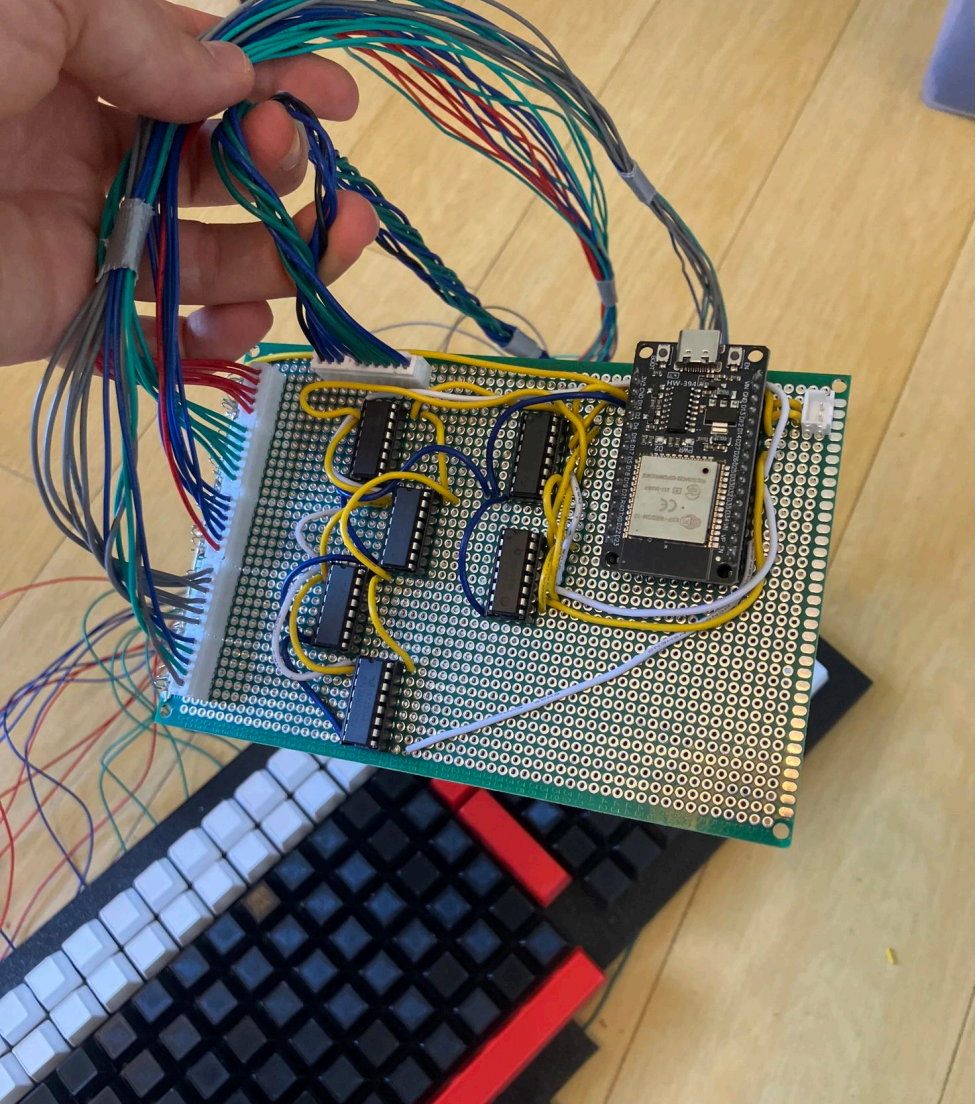


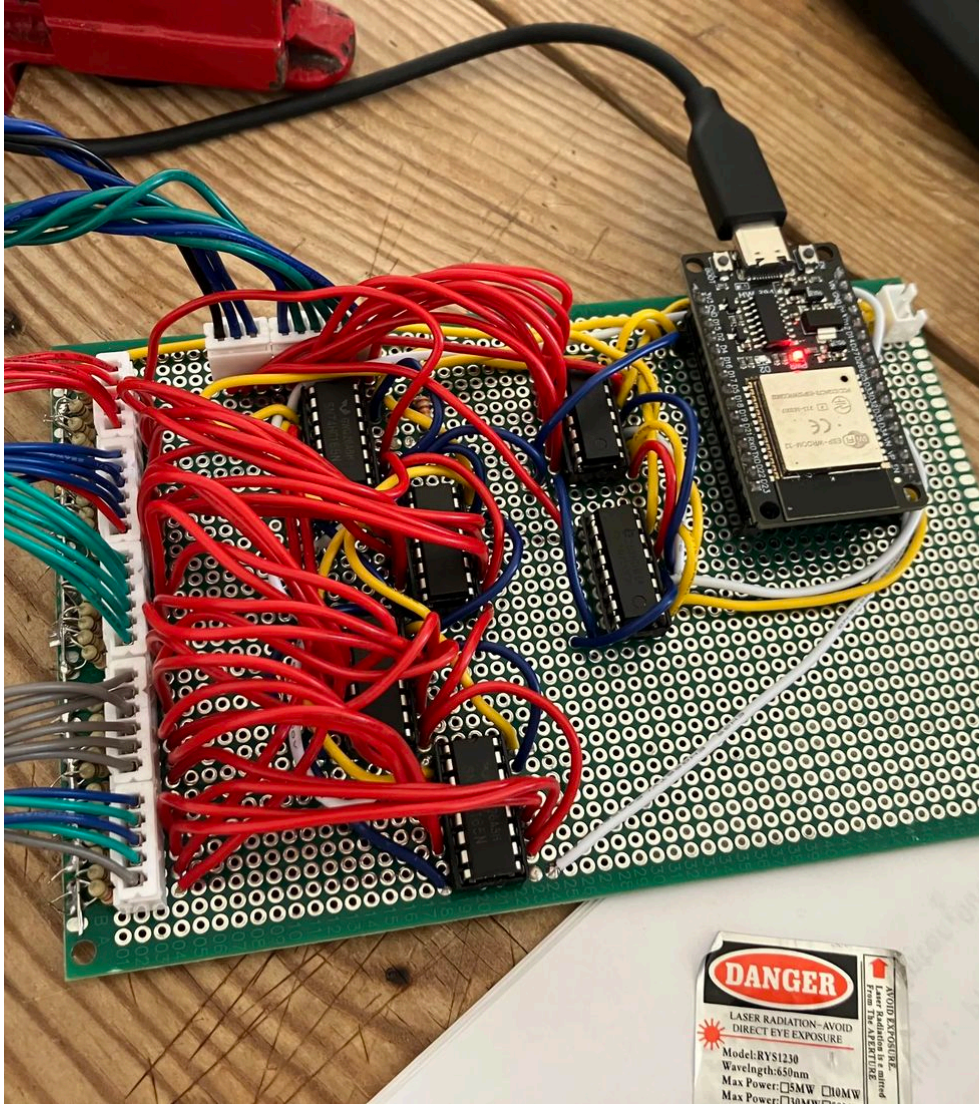
font_size_gauge / NeedleGauge

font_size_joystick / Joystick








DANGER
LASER RADIATION-AVOID DIRECT EYE EXPOSURE
Model:RYS1230
Wavelngth:650nm
Max Power:□5MW □10MW



STAND ALONE CONFIGURATION MODE MENU TABLE

NO	LED1(ON)	LED2(BLINK)	PRINTER OPERATION
1	GREEN	GREEN	PRINT CONFIGURATION INFO.
2	GREEN	ORANGE	PRINT FILE LIST
3	GREEN	RED	FACTORY RESET
4	ORANGE	GREEN	GAP – CALIBRATION MODE
5	ORANGE	ORANGE	B/M – CALIBRATION MODE
6	ORANGE	RED	START DATA – DUMP MODE
7	RED	GREEN	DEMO MODE

XPM	: DISABLE
REFLECT SENSOR	: H:2.06V, L:1.03V, D:1.03V, T:1.55V
TRANSMIT SENSOR	: H:2.03V, L:0.77V, D:1.26V, T:1.48V
CAL CHECK UNIT	: 400

CONFIGURATION CONTROL STATUS

PRINT SPEED	: COMMAND
PRINT DARKNESS	: COMMAND
PRINT WIDTH	: COMMAND
PRINT OFFSET	: COMMAND
TEAR OFF	: COMMAND

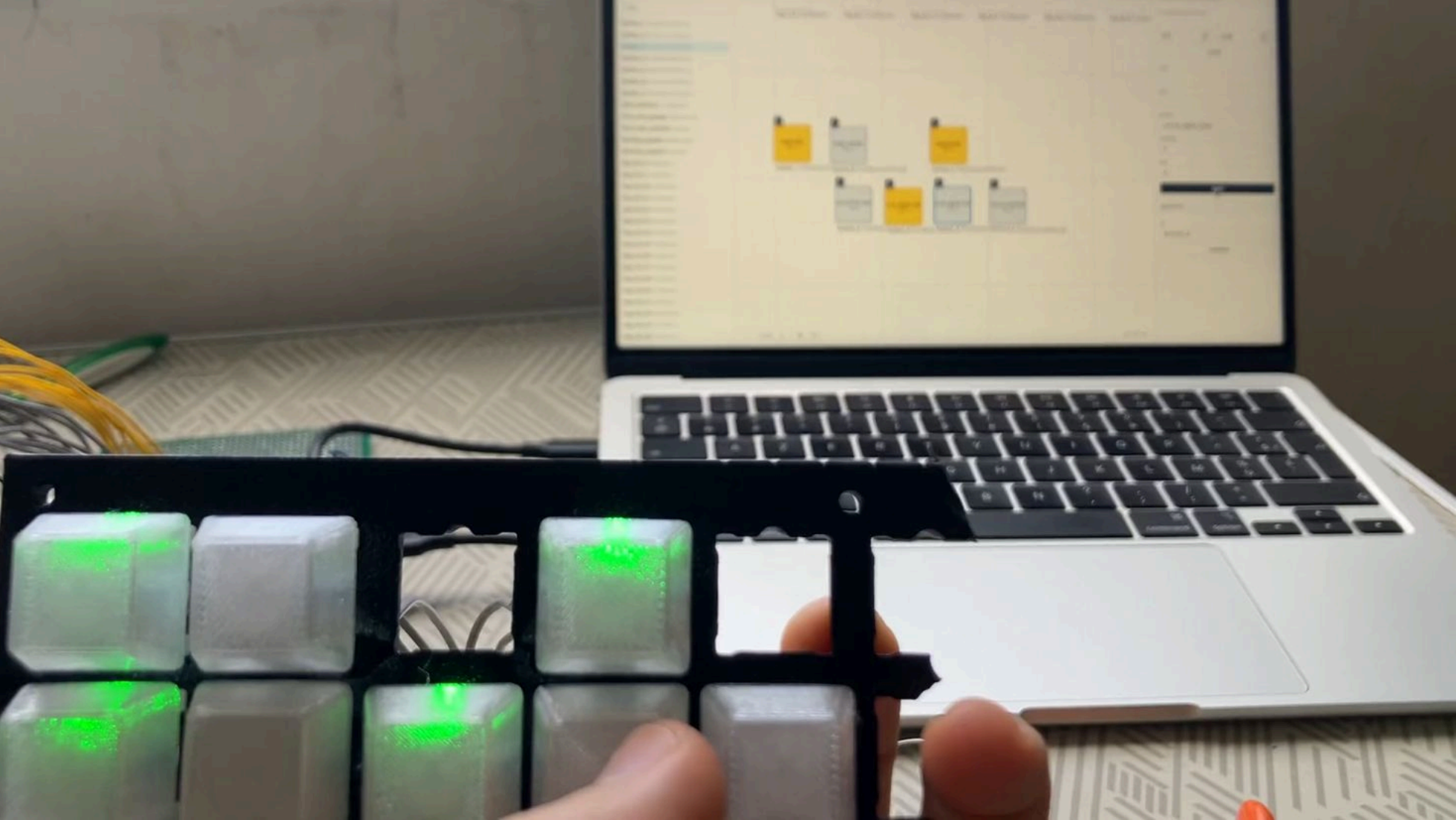
BIXOLON XD5 – 43d CONFIGURATION

FIRMWARE VERSION	: V01.40 STD 101722
EMULATION	: SLCS & BPL – Z & BPL – E
PRINT DARKNESS	: 20/30
PRINT SPEED	: 4 IPS
TEAR OFF VALUE	: + 000 DOT
TEAR OPTION	: CUTTER
PRINT OFFSET	: + 0000
POWER – UP ACTION	: NO ACTION
MEDIA TYPE	: WEB PAPER

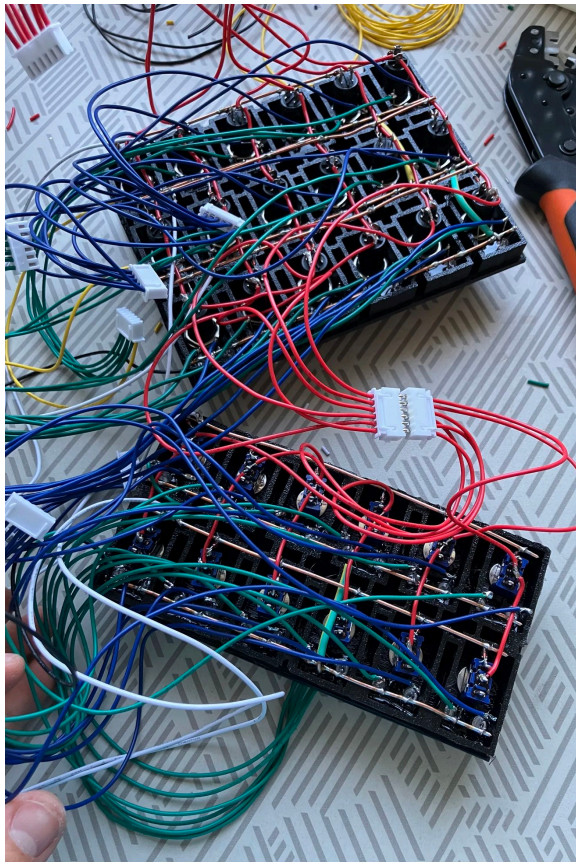
: 601 DOT
: 1232 DOT
: 115200
: NONE
: 8
: 1
: HARDWARE
: BEGWWRKA24020051
: CP – 850 (SBE. USA – 1 CHARSE
: WCP1252 – LATIN1
: U.S.A
: 1/1/11
: 0:0
: DISABLE

: COMMAND
: COMMAND

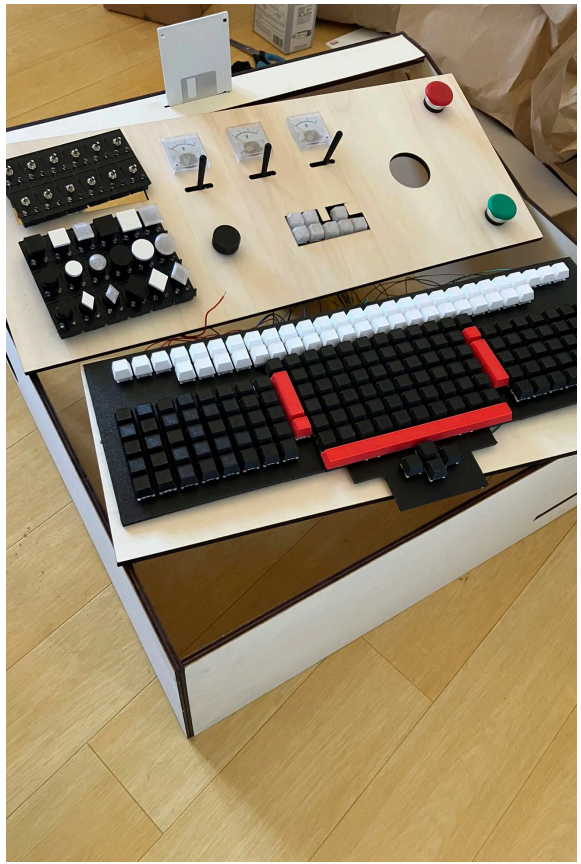




Flashing slide next








ISITA
Maëva
cœur
téte

Where do we put complexity ?

- Hardware as simple as possible (provide inputs/outputs)
- Firmware as simple as possible (send events, get commands)
- All logic high-level in the Elixir host

How do we discover and refine our design ?

- Simulate each component in Elixir
- Keep the logic as pure as possible
- Compose the system from components in a control panel
- Move them around, experiment with different types of interactions and effects
- Advance the build, it helps refine the code
- Refine the code, it helps advance the build

Firmware doesn’t know its purpose

```
void app_setup() {
  serial_link::begin(proto::kBaud, proto::kModuleId);
  key_spi.begin(kSck, kMiso, kMosi, -1);
  drive.begin();
  sense.begin();
  keys.set_settle(settle);
}

void app_loop() {
  serial_link::poll(apply_command);
  const unsigned long now = millis();
  if (now - last_scan_ms < kScanIntervalMs) {
    return;
  }
  last_scan_ms = now;
  keys.scan(on_key_event);
}
```

Each control type implements `Component`

```elixir
defmodule Hardware.Component do

  @callback new(opts) # The data that represents me
  @callback events() :: list(atom()) # What I react to
  @callback effects() :: list(atom()) # What I do on the world
  @callback handle_event(state, event, payload) :: # How do I react to an event, what do I emit ?
              {:ok, new_state} | {:ok, new_state, [{atom(), term()}]}
  @callback handle_effect(id, effect, payload, old_state, new_state) :: :ok
  @callback input_binding(state) :: term() | nil # How to map me to hardware ?

  @callback editable_fields() :: keyword(atom()) # How can I be configured ?
  @callback dimensions() :: {number(), number()} # How do I display ?
```

```elixir
defmodule Hardware.Components.PushButton do
  @behaviour Hardware.Component
  defstruct [:pushed, :char, :shift, :action, :red]

  def new(opts) do
    %__MODULE__{ pushed: false,
      char: Keyword.get(opts, :char, nil),
      shift: Keyword.get(opts, :shift, nil),
      action: Keyword.get(opts, :action, nil),
      red: Keyword.get(opts, :red) == true
    }
  end

  def editable_fields(), do: [char: :string, shift: :string, action: :string, red: :boolean]
  def events(), do: [:key_up, :key_down, :set_pushed]

  def handle_event(state, :key_down, _),
    do: {:ok, %{state | pushed: true}, [{:key_down, meaning(state)}]}

  def handle_event(state, :key_up, _),
    do: {:ok, %{state | pushed: false}, [{:key_up, meaning(state)}]}
end
```

```
iex> PushButton.new(char: "a", shift: "A")
      |> PushButton.handle_event(:key_down, nil)

{:ok, %PushButton{pushed: true}, [{:key_down, %{char: "a", shift: "A", action: nil}}]}

# {:ok, new state, normalized events}
```

Some controls types implement `Output`

```elixir
defmodule Hardware.Output do
  @callback handle_transition(old :: struct(), next :: struct(), opts :: keyword()) :: :ok
end
```

```elixir
defmodule Hardware.Components.PushButtonWithLed do
  @behaviour Hardware.Component
  @behaviour Hardware.Output

  alias Bbparty.Bus.LedShadow
  alias Bbparty.Keyboard.Action

  defstruct pushed: false, lit: false, char: nil, shift: nil, action: nil, module: nil, led: nil
  def events(), do: [:key_up, :key_down, :set_pushed, :set_lit]

  @impl Hardware.Output # Impure, affects the world
  def handle_transition(%__MODULE__{} = _old, %__MODULE__{} = new, _opts) do
    LedShadow.put(new.module, new.led, new.lit)
    :ok
  end
end
```

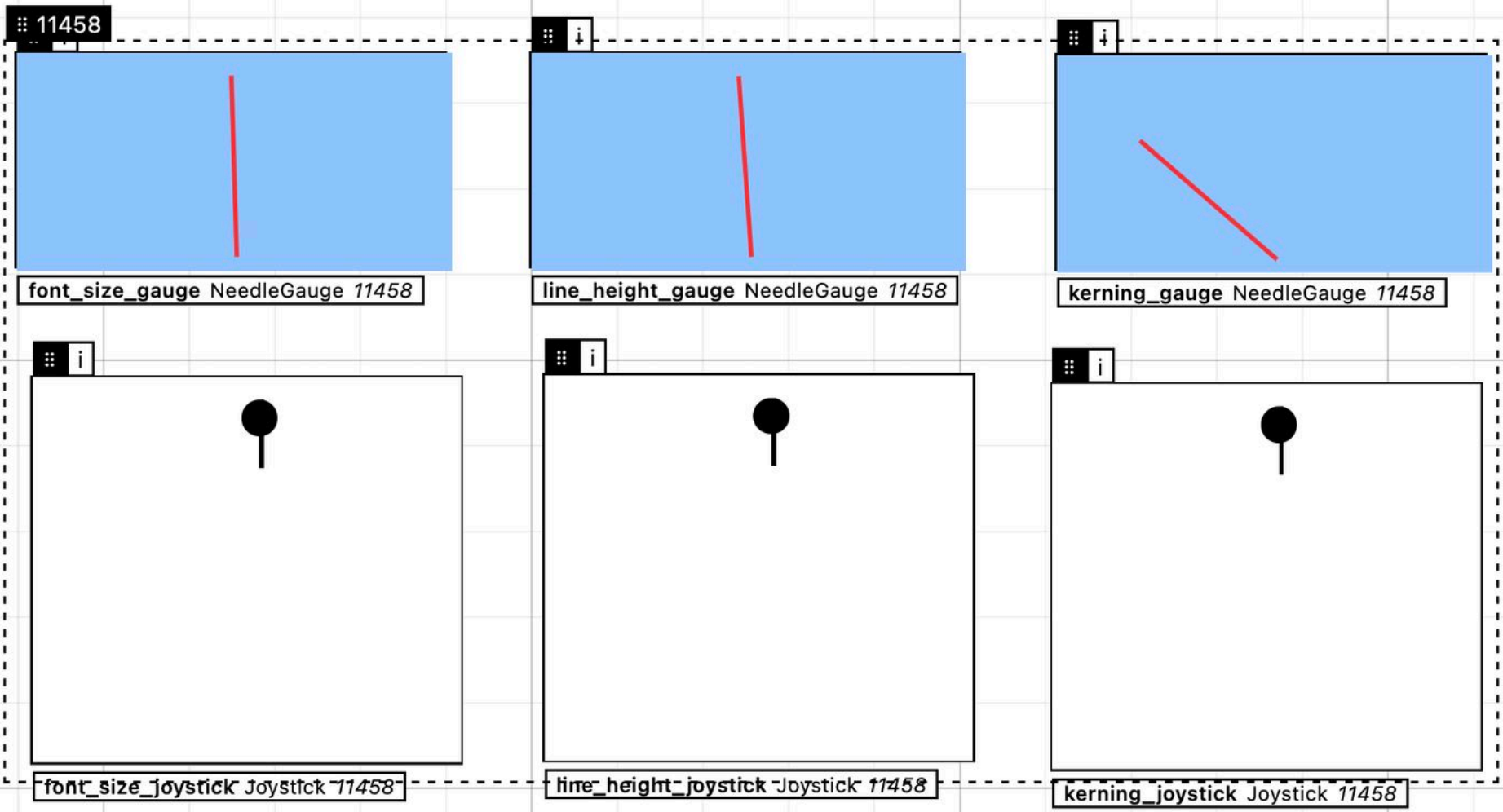
Every control type is capable of being rendered

```
defmodule Hardware.Components.NeedleGauge do
  @behaviour Hardware.Component
  @behaviour Hardware.Output

  defstruct progress: 50, module: nil, gauge: nil, duty_min: 0, duty_max: 255
  ...
```

```
defmodule Hardware.Components.NeedleGauge.Display do
  use Phoenix.Component

  attr :data, :map
  def render(assigns) do
    ~H"""
    <div class="w-48 h-24 bg-blue-300 relative">
      <div
        class="needle w-[2px] bg-red-500 h-[80px] left-[50%] top-[10px] absolute origin-bottom"
        style={"transform: rotate(#{compute_rotation(@data.progress)}deg)"}
      >
      </div>
    </div>
    """
```


11458
font_size_gauge NeedleGauge 11458
line_height_gauge NeedleGauge 11458
kerning_gauge NeedleGauge 11458
font_size_joystick Joystick 11458
line_height_joystick Joystick 11458
kerning_joystick Joystick 11458

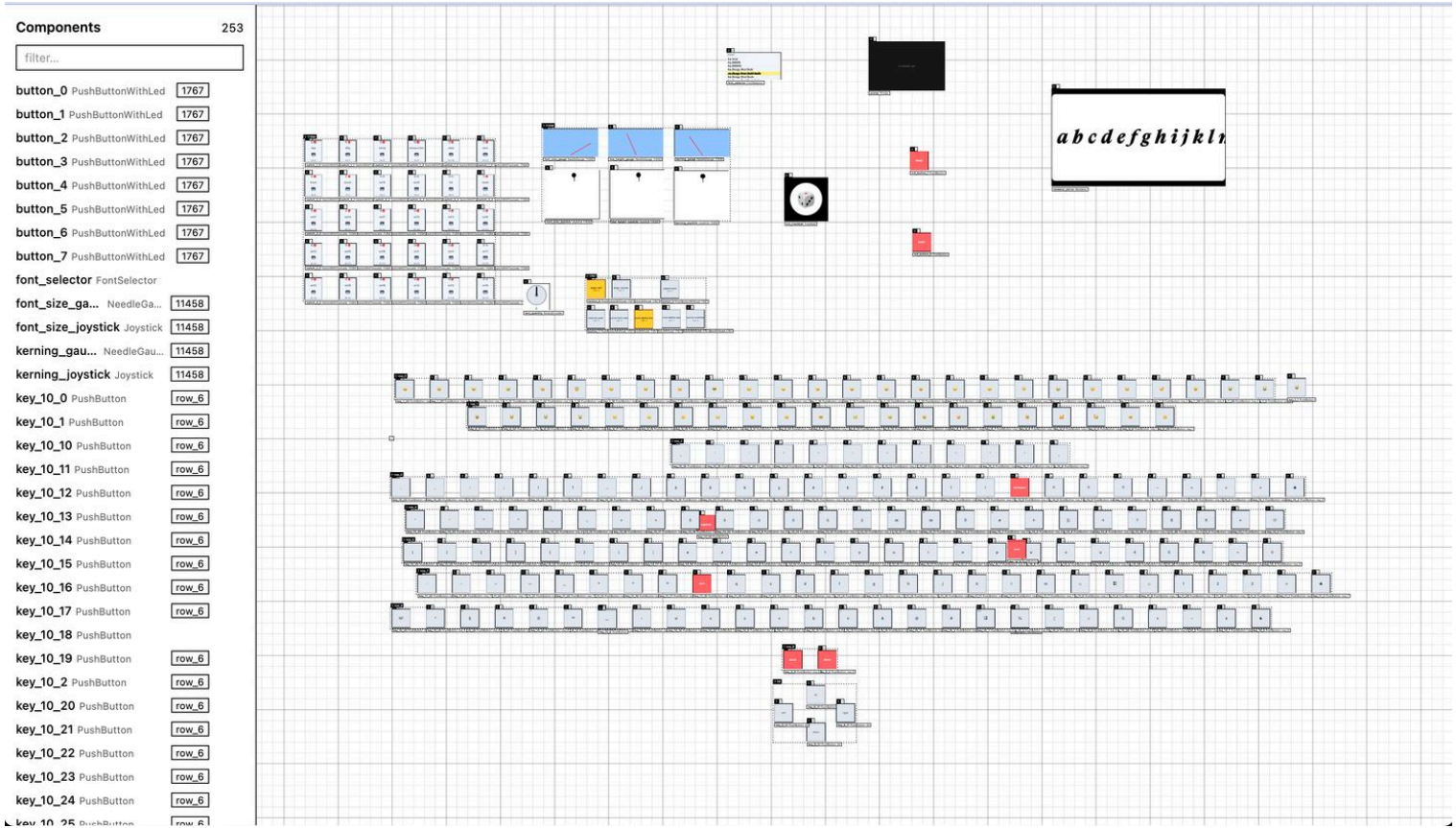
Behavior = type + instance, a `PhysicalComponent` GenServer

```
[info] Start switch_4_5 via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_switch_4
[info] Start text_quantity via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_text_
[info] Start renderer_mirror via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_ren
[info] Start m4_trackball via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_m4_tra
[info] Start font_size_joystick via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_
[info] Start kerning_gauge via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_kerni
[info] Start line_height_gauge via {:via, Registry, {Bbparty.ComponentsRegistry, :PhysicalComponent_l
```

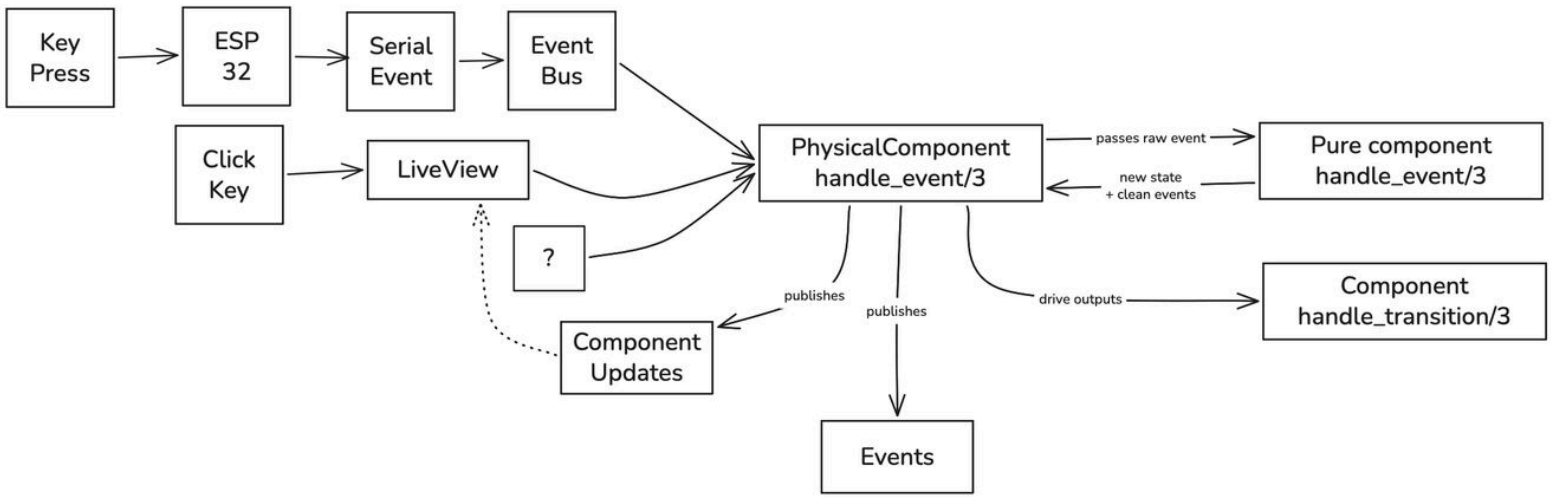
```elixir
defmodule Bbparty.Layout.ComponentInstance do
  use Ecto.Schema
  schema "component_instances" do
    field :component_id, :binary
    field :module, :binary
    field :attrs, :map
    field :position, :map
    field :group_id, :binary

    timestamps(type: :utc_datetime)
  end
```

Panel visit : instantiated components = routing from the world to logic



How events enter the system



Event loop (1/2)

- An event enters from hardware
- Its `PhysicalComponent` is resolved from a registry
- It enters through `handle_event/3` (or `feed_input/3` for high frequency events)
- The pure `handle_event/3` is called -> `{:ok, state, emits}` (emits are normalized events)
- `PhysicalComponent` publishes :
  - `{:component_changed, id, new_state}` for LiveView
  - and `{:component_event, id, event, payload}` for the core
- Impure effects (randomness, time) are injected by `RenderStateServer`
- `RenderState.apply` folds the event on the state
  - ```
    {new_state, effects} = do_apply(state, event, layout)
    {new_state, Projections.propagate(state, new_state), effects}
    ```
  - Dependencies of the changes are resolved and become commands
    ```

Event loop (2/2)

- `{render, commands, effects} = RenderState.apply/3`
- `RenderStateServer` dispatches commands (adjust your gauge…) to relevant `PhysicalComponent`s
- Components `handle_event/3` the commands and update in LiveView
- Output-enabled `PhysicalComponent` run `handle_transition/3` and impurely drive outputs
- Any cast effects on the world (printing) are run
- The actual rendering updates

A key press from the hardware :

```
PhysicalComponent.handle_event(id, :key_down, nil)
```

A key press from liveview :

```
PhysicalComponent.handle_event(id, :key_down, nil)
```

A key press from elsewhere :

```
PhysicalComponent.handle_event(id, :key_down, nil)
```

RenderState : easy high-level view of the logic + dedicated pure logic modules

```
def apply(state, event, layout) do
  {new_state, effects} = do_apply(state, event, layout)
  {new_state, Projections.propagate(state, new_state), effects}
end

defp do_apply(state, {id, :nudge, value}, _l), do: {Sizings.nudge(state, id, value), []}

defp do_apply(state, {_id, :nudge_quantity, chars}, _l),
  do: {TextSource.nudge_quantity(state, chars), []}

defp do_apply(state, {_id, :toggle_feature, tag}, _l),
  do: {Font.toggle_feature(state, tag), []}

defp do_apply(state, {_id, :set_feature, {tag, on?}}, _layout),
  do: {Font.set_feature(state, tag, on?), []}

defp do_apply(state, {_id, :select_font, font}, _layout),
  do: {Font.select(state, font), []}
```

Projections issue dependent update commands

```
def projections do
  %{
    ...
    text_quantity: &quantity_commands/1,
    alignment: &alignment_lamps/1,
    transform_toupper: &uppercase_lamp/1,
    text_source: &source_lamps/1,
    cumulative_features: &feature_active_commands/1,
    selected_font: &font_commands/1
  }
end

defp alignment_lamps(alignment) do
  [
    {{:action, :align_left}, {:set_lit, alignment == :left}},
    {{:action, :align_center}, {:set_lit, alignment == :center}}
  ]
end
```

Dedicated pure modules handle the detail of event application

```
defmodule Bbparty.RenderState.TextSource do
  @sources %{
    source_user: :user,
    source_font_name: :font_name,
    source_alpha_upper: :alpha_upper,
    source_alpha_lower: :alpha_lower,
    source_numbers: :numbers
  }

  def nudge_quantity(state, chars),
    do: %{state | text_quantity: clamp(state.text_quantity + chars, 0, TextToWrite.count())}

  def effective_text(%RenderState{text_source: :font_name, selected_font: font}), do: font || ""
  def effective_text(%RenderState{text_source: :alpha_upper}), do: @alpha_upper
  def effective_text(%RenderState{text_source: :alpha_lower}), do: @alpha_lower
  def effective_text(%RenderState{text_source: :numbers}), do: @numbers
  def effective_text(%RenderState{text_quantity: n}) when is_integer(n) and n > 0, do: TextToWrite.ta
  def effective_text(%RenderState{text: text}), do: text

  def overridden?(%RenderState{text_source: source}) when source != :user, do: true
  def overridden?(%RenderState{text_quantity: n}), do: is_integer(n) and n > 0
```

Impure steps of "align left" pressed

- I press the physical key
- ESP32 scans keys and sends `<<0x82, 6, 1>>` (key event, line 6, down)
- Transformed to `{:frame, 0x02, <<0x82, 6, 1>>}` routes to a Module2Dispatcher
- Module2Dispatcher calls `PhysicalComponent.handle_event(:button_6, :key_down, nil)`
- `button_6` found in the registry, `%PushButtonWithLed{action: "align_left", module: 2, led: 2}`.
  - `PushButtonWithLed.handle_event/3` -> `{:ok, %{state | pushed: true}, [{:key_down, %{char: "", shift: "", action: "align_left"}}]}`.
    - broadcast `{:component_event, :button_6, :key_down, meaning}`
- `maybe_output/3` → `PushButtonWithLed.handle_transition/3` → `LedShadow.put(2, 2, true)`
- `LedShadow` maintains a bit mask per board that has LEDs
- Next 20ms tick LEDs are flushed -> `Module2.set_leds(<<0b0000_0100>>)`

Build prooess

- Design on paper
- Implement the event + render pure loop
- Implement the first components and the control panel / render liveview
- Prepare the boards, generate the firmwares ← a few recurrent bugs despite the simplicity, blamed my soldering for firmware bugs
- Correct the firmwares manually
- Let it run for days unattended at most points of the process
- AI helped with :
  - subsequent components and helpers ← muddied my design too much for my taste
  - very niche parts of the effects ← efficient but bruteforce-y
  - generate paper task-lists for soldering ← very helpful, no regrets
  - pimping the digital twin UI/UX ← same
- today : Refactor in progress of the non-human parts for the next exhibitions

What I would change

- Better distinction between event types (world event, internal command) and their handlers
- No distinction between outputs and effects : my initial reason to separate them did not materialize
- A connected Livebook instead of the codebase for all one-off commissioning scripts

Benefits

- Very late resolution of handlers
- Agents can drive the system from the events without the hardware → demo
- Live-reconfigurable installation
- No coupling between hardware and logic
- Other transports are easy to add
- Liveview isn't the app but a simple renderer

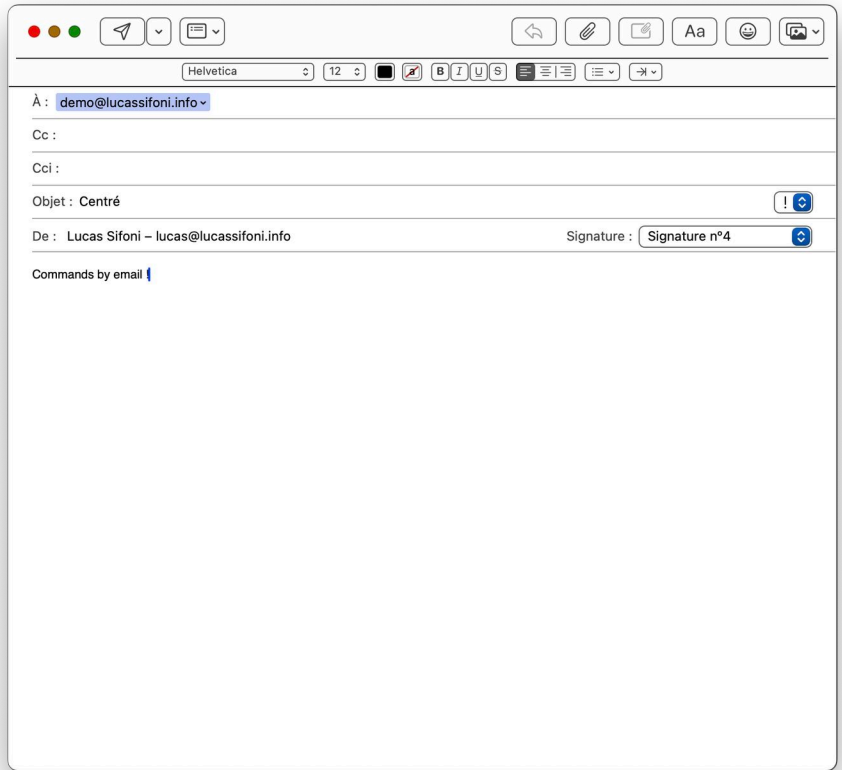
There was no simulator ! → The simulator is the system

Beyond hardware

Why manage state in a similar way for any* app

- Easily migrate to another framework
- Build liveview states in scripts and tests
- Inform of pure function heads and docs via MCP, interpret lists of commands, build state
- Universal UIs + thin rendering layer

Alternative transports : I mean it



contact@lucassifoni.info

20 minutes is short : code, more photos, french version : https://pushin.eu/lsifoni/codebeam26

Personal website -> lucassifoni.info
My SaaS company -> alzo.fr
My side-project -> chantepierre-optique.fr

