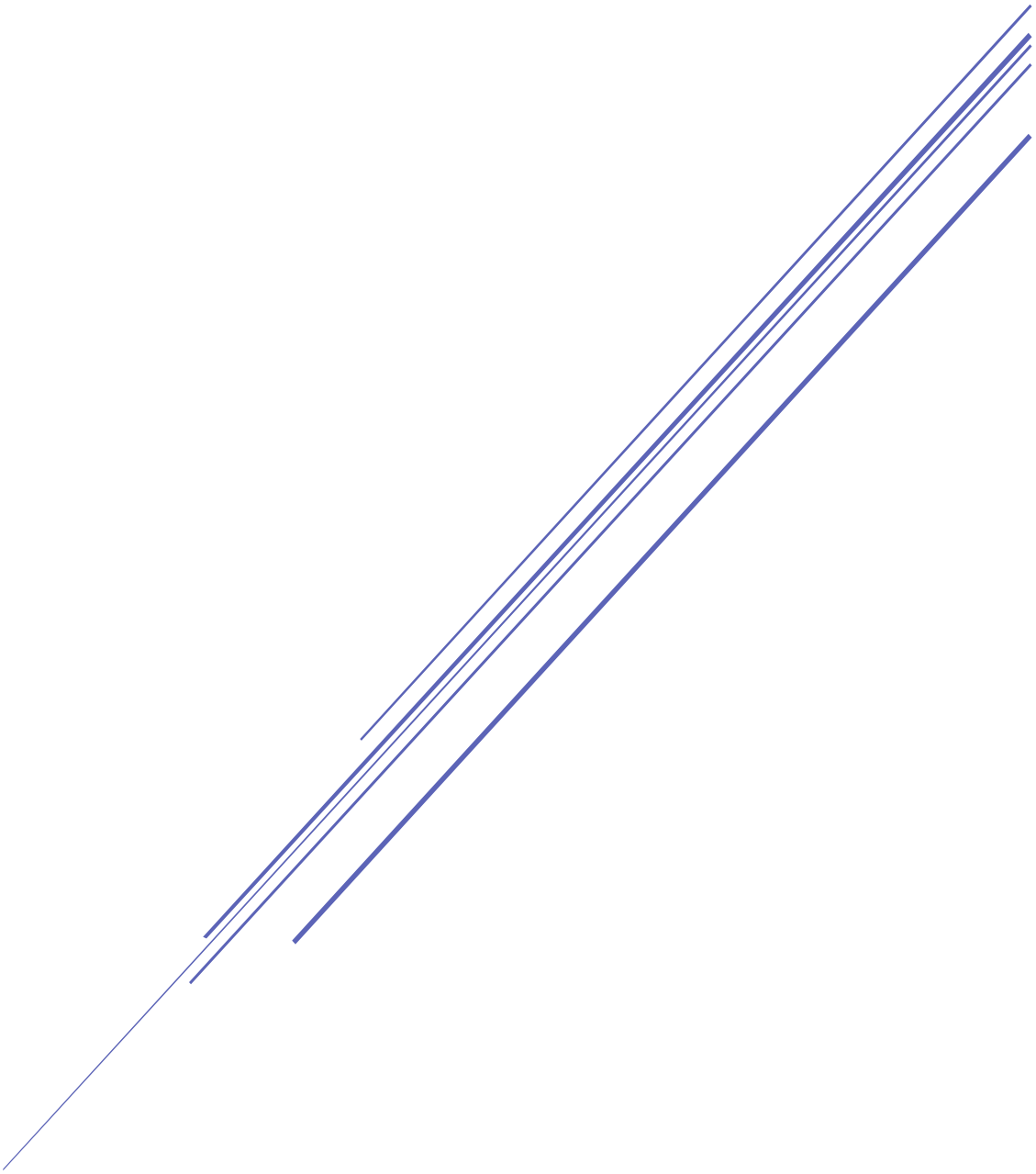
µLIFE

Development journal



Matthieu MICHEL
January, 2025

1 Table of Contents

1.1 Tables of figures

2 Project overview

2.1 Objective

I recently made an implementation of the Game of Life from John Conway in C++ for a school project and I found it interesting, I was mainly amazed by how complex can such a simple get.

I then thought to myself I had to make some hardware related to this game. When I was watching bitluni's videos, it popped in my mind like an obviousness, I had to make a keychain with the game of life on it! This way not only will my key ring become way cooler, but I'll have a great conversation starter and work to show when questioned about my hobby.

2.2 Scope

The project should consist of an LED matrix with simple controls allowing the Game of Life to run on it. It will display preprogrammed patterns, and the user should also be able to display a randomized grid, also known as "soup". It will also have a cover to protect from keys and other items. Speed controls will be available and there may be an Easter egg.

The project will make use of charlieplexing. Charlieplexing is a technique used to access many LEDs with few pins using tri-state logic. This will allow us to control our matrix with a few I/O pins from our microcontroller.

2.3 Constraints

The project shall meet the following criteria:

- The product should be no larger than 3x4cm.
- The LEDs should not blind the user.
- The BOM shall not exceed €30.
- The product should have a protective case or cover.
- The product should not feel hot to touch.

2.4 Timeline

This project does not have any time constraints.

3 Planning

3.1 Requirements

3.1.1 Functional requirements

The functional requirements are actions the system does. They specify functions, features or tasks the product must perform. They are often described as clear tasks.

The functional requirements are the following:

- The product must be USB-C powered.
- The onboard MCU should be able to be reprogrammed.
- The product must have physical controls like buttons.
- The product must include speed control.

3.1.2 Non-functional requirements

Non-functional requirements describe how well the system performs, operates, or adheres to certain constraints. They are often quality attributes or external limitations. They often define standards rather than tasks.

For our project, they are as such:

- A LED should not consume no more than ~~8mA~~ 20 mA (peak).
- The display should be able to reach 60 fps.
- The display should be visible in sunlight.
- The whole system's current consumption should not exceed 250 mA.
- The enclosure must withstand minimal scratches.
- The enclosure must be transparent.
- The product must not directly expose the user to lead solder or any other toxic compounds.
- The PCB must be designed to allow for single-sided assembly.
- The circuit must make use of the least amount of unique parts to reduce assembly costs.

3.1.3 Hardware requirements

Hardware requirements are requirements based on needed parts for the project.

We have the following parts:

- STM32F030R8T6, for large IO count.
- 0402 white LEDs, as small LEDs are a necessity here.

3.2 Initial design

For the design of this project, I thought about making a game console-like circuit, with power at the bottom of the board, followed by the MCU and two buttons for controlling it. Above we will have the LED matrix in all its glory.

Here is the blocks diagram:

Figure 1: Blocks diagram of the initial design

I find this architecture easy, and it should make layout and routing rather easy, except for that 400 LEDs matrix…

3.3 Implementation plan

The plan is here to first make the LED matrix with headers and use a NUCLEO-F030R8 board to check the feasibility of the project and how well frames can be displayed on the matrix.

The second step is to make the final product, putting down a STM32F030R8 with a crystal on the board as well as power and buttons, giving us the full fledge product.

Lastly, we'll design a cover for the keychain to protect it against external inconveniences.

4 Prototypes

4.1 Prototype N°1

4.1.1 Goal

The objective of this first prototype is to check the feasibility of our concept by checking if we can successfully display things on our display.

4.1.2 Design & diagrams

To have numerous LEDs on a single board on only 21 GPIOs, we're going to make use of Charlieplexing. Charlieplexing is a technique making use of tri-state logic to control a lot of I/O rather easily.

We first put LEDs between each pair of GPIO, excluding pairs where $a = b$. For example, for 3 pins we would have: 1/2, 1/3, 2/3, 2/1, 3/1, 3/2. We then utilize the three different possible states of our GPIO pins: high, low, and high-Z (or disconnected). We can put every pin in high-Z, so the MCU won't be trying to drive the pin low or high, effectively disconnecting it from the circuit. At this point we choose two pins, set one high, and the other low and we have a working LED.

This way we can control, with N the number of GPIOs, $N(N-1)$ with Charlieplexing as opposed to only $\frac{N^2}{4}$ LEDs with standard multiplexing. We can even find the number of required pins the number of LEDs is known with the following equation: $n = \lceil 1 + \sqrt{L} \rceil$. In our case we then need $1 + \sqrt{400} = 21$ pins.

More can be read on Wikipedia's article on the matter (link in the references).

To generate our LED matrix, we're obviously not going to do it by hand. Fortunately for us, psychogenic on GitHub made a script to generate them for us. After use, we end up with this:

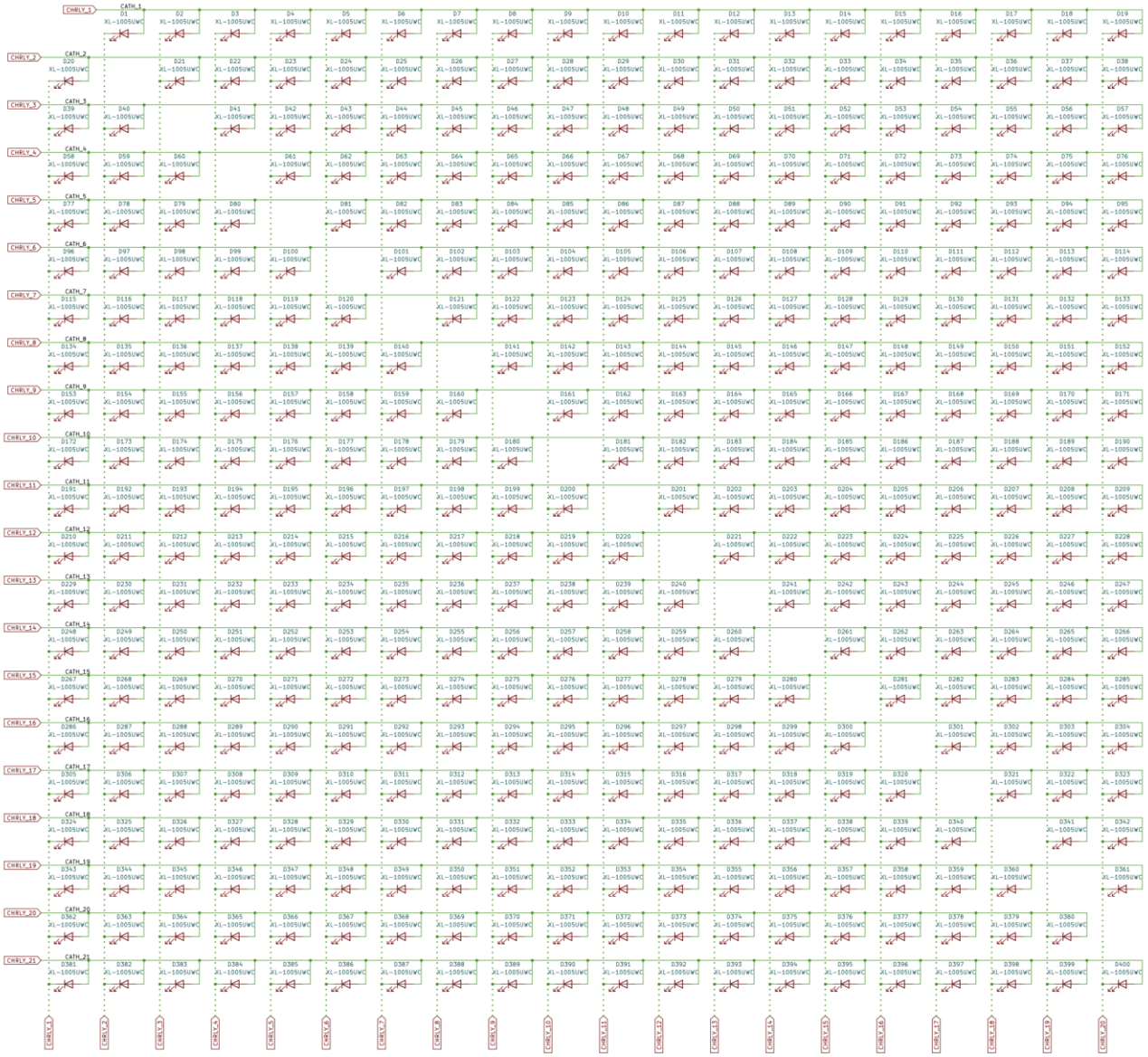


Figure 2: Prototype N°1 – Charlieplexed LED matrix

We will mention the presence of 21 rows, since the diagonal is absent, we must add another row and shift them up to get a full square matrix.

I then added two female headers to access our matrix, one with 100 Ω and the other with 220 Ω resistors to test different series resistors.

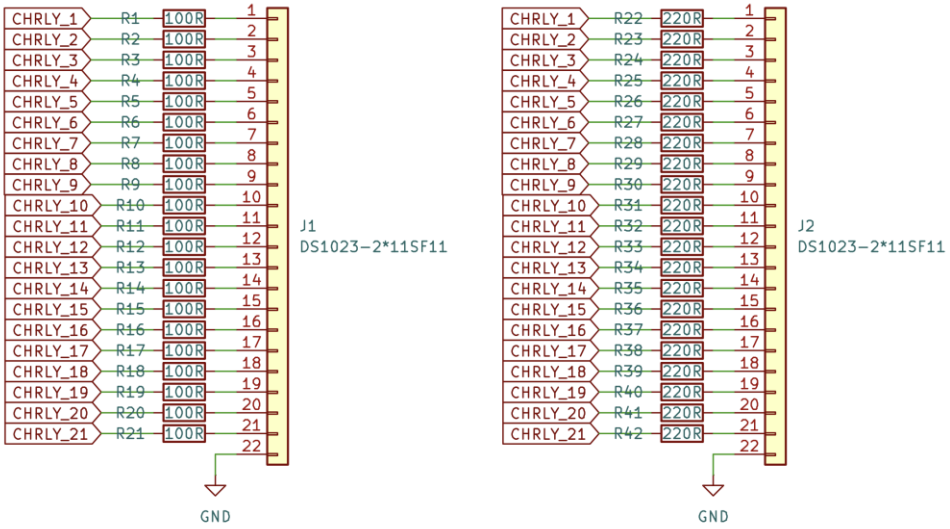


Figure 3: Prototype N°1 – 2x11 female headers

Those resistors are calculated for 2 and 1mA respectively.

PCB layout will not be discussed here as it is not relevant.

4.1.3 Assembly notes

I was definitely not going to assemble this myself, as the chance to reverse an 0402 LED is very high (and the chance of me going insane is non-negligeable too). I made use of JLCPCB's assembly service. With a good coupon, it came down to €12.58 shipped for two assembled boards and the leftover PCBs.

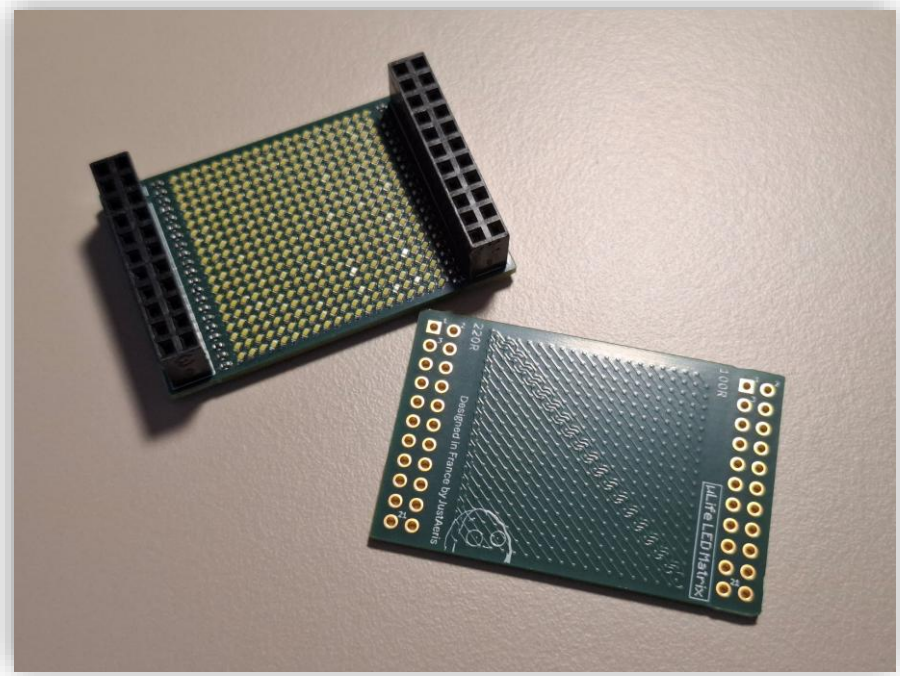


Figure 4: Prototype N°1 – Assembled boards

Boards came clean, white LEDs have a small yellow tint on the lens, but it may not be considered an issue.

4.1.4 Testing

To test our matrix, we're going to wire it up to our NUCLEO-F030R8 according to the table below. No specific thinking has gone into the selection of used pins.

Header pin	NUCLEO pin	Header pin	NUCLEO pin
2	B1	1	B0
4	B3	3	B2
6	B5	5	B4
8	B7	7	B6
10	B9	9	B8
12	B11	11	B10
14	B13	13	B12
16	B15	15	B14
18	C1	17	C0
20	C3	19	C2
22	NC	21	C4

Figure 5: Prototype N°1 – Headers to NUCLEO-F030R8 pinout

Note: this is 1:1 to the header itself, indifferent of each side.

4.1.5 Software implementation

Software-side, I implemented first a function to set back all inputs to reset state (equivalent to input mode) using registers for faster operations.

```
void Charlie_Clear() {
    // Reset GPIOB
    GPIOB->MODER = 0x00000000;    // Set all pins to input (reset state)
    GPIOB->OTYPER = 0x00000000;   // Set all pins to push-pull (default)
    GPIOB->OSPEEDR = 0x00000000;  // Set all pins to low speed
    GPIOB->PUPDR = 0x00000000;    // No pull-up/pull-down resistors

    // Reset GPIOC
    GPIOC->MODER = 0x00000000;
    GPIOC->OTYPER = 0x00000000;
    GPIOC->OSPEEDR = 0x00000000;
    GPIOC->PUPDR = 0x00000000;
}
```

Code 1: Prototype N°1 – Charlieplexing clear function

Note: this resets the whole GPIO B and C ports, meaning we can't use C5 to 15 right now.

We then have a simple method, taking in a GPIO port, pin and number and from that, setting the pin type as well as its state:

```
void Charlie_SetPinState(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, int state)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    if (state == 1) { // Set as Source (HIGH)
        GPIO_InitStruct.Pin = GPIO_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
        HAL_GPIO_WritePin(GPIOx, GPIO_Pin, GPIO_PIN_SET);

    } else if (state == 0) { // Set as Sink (LOW)
```

```
        GPIO_InitStruct.Pin = GPIO_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
        HAL_GPIO_WritePin(GPIOx, GPIO_Pin, GPIO_PIN_RESET);

    } else { // Set as High-Z
        HAL_GPIO_DeInit(GPIOx, GPIO_Pin);
    }
}
```

Code 2: Prototype N°1 – Function to set a pin state

And this is where the fun begins. Because now, we must find a way to address each LED and since the LEDs' pins don't follow a linear pattern, the only (and sad) way to address them with (x, y) is with a lookup table. And this is where I'm glad to be in 2025 and have AI tools because with Claude AI, I managed to get my lookup table with rather low efforts:

```
// LED Matrix Position to GPIO Pin Mapping
// Format: {HighGPIO, HighPin, LowGPIO, LowPin} representing (high,low) charlieplex pairs
typedef struct {
    GPIO_TypeDef* highPort;
    uint16_t highPin;
    GPIO_TypeDef* lowPort;
    uint16_t lowPin;
} CharliePinMapping;

CharliePinMapping LED_Matrix[][20] = {
        // Row 0 - Pattern: (1,2) to (1,20), then (2,1)
        {
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_1},     // (1,2)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_2},     // (1,3)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_3},     // (1,4)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_4},     // (1,5)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_5},     // (1,6)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_6},     // (1,7)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_7},     // (1,8)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_8},     // (1,9)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_9},     // (1,10)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_10},    // (1,11)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_11},    // (1,12)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_12},    // (1,13)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_13},    // (1,14)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_14},    // (1,15)
                {GPIOB, GPIO_PIN_0, GPIOB, GPIO_PIN_15},    // (1,16)
                {GPIOB, GPIO_PIN_0, GPIOC, GPIO_PIN_0},     // (1,17)
                {GPIOB, GPIO_PIN_0, GPIOC, GPIO_PIN_1},     // (1,18)
                {GPIOB, GPIO_PIN_0, GPIOC, GPIO_PIN_2},     // (1,19)
                {GPIOB, GPIO_PIN_0, GPIOC, GPIO_PIN_3},     // (1,20)
                {GPIOB, GPIO_PIN_1, GPIOB, GPIO_PIN_0}      // (2,1)
        },
        // ...
};
```

Code 3: Prototype N°1 – Part of the lookup table

And this is (ironically) the hardest part done. Now a little function taking an item of our lookup table and do the magic:

```
void Charlie_SetLED(int x, int y) {
    // Get the corresponding high and low GPIOs from the table
    GPIO_TypeDef* highPort = LED_Matrix[x][y].highPort;
    uint16_t highPin = LED_Matrix[x][y].highPin;
    GPIO_TypeDef* lowPort = LED_Matrix[x][y].lowPort;
```

```
    uint16_t lowPin = LED_Matrix[x][y].lowPin;

    Charlie_Clear();

    // Set the high and low pins
    Charlie_SetPinState(highPort, highPin, 1); // Set highPin as HIGH (source)
    Charlie_SetPinState(lowPort, lowPin, 0);   // Set lowPin as LOW (sink)*/
}
```

Code 4: Prototype N°1 – Pin setting function

And now we can control our matrix using (x, y) coordinates like a regular grid.

Now onto the actual displaying part, since only one LED can be on at a time we need to cycle through all the LEDs fast. To achieve a 60 Hz refresh rate, we need to update our whole display every $\frac{1}{60} = 0.017\ s = 17\ ms$ and since we have 400 LEDs, we need to spend $\frac{0.017}{400} = 42.5 \times 10^{-6}\ s = 42.5\ \mu s$ per LED.

Knowing this, we can set up a timer and an interrupt to kick in every 42.5 µs. To accomplish this, we set up TIM3 on a 2 MHz clock using a prescaler of 23, originating from this formula:

$$Prescaler = \frac{System\ clock}{Desired\ timer\ clock} - 1$$

Applied to our case:

$$\frac{48 \times 10^6}{2 \times 10^6} - 1 = 24 - 1 = 23$$

This gives us a timestep of 0.5 µs.

A timer is simply a counter going up by one every time it ticks, we must then determine the ARR, for AutoReload Register, also known as the counter period. When our timer reaches this value, our interruption will fire, and it will reset the counter back to zero. This effectively means that our timer's interrupt will fire every $ARR \times 0.5\ \mu s$ in our case. We already determined that we need to update a LED every 42.5 µs, dividing this timestep by our tick length, 0.5 µs, will give us our ARR: $\frac{42.5}{0.5} = 85$. We can easily, based on the ARR and prescaler, find to what refresh rate our display is on using the formula:

$$f_{display} = \frac{f_{SysClock}}{ARR \times (Prescaler + 1) \times N_{LEDs}}$$

Using the values calculated above, we are given a 58 Hz refresh rate, which is enough.

We also need to enable the TIM3 global interrupt in the NVIC settings. After doing, we can simply create a global variable, a 2D array, representing our display and in the interrupt, update each LED according to this 2D array.

```
void TIM3_IRQHandler(void)
{
  /* USER CODE BEGIN TIM3_IRQn 0 */

  /* USER CODE END TIM3_IRQn 0 */
  HAL_TIM_IRQHandler(&htim3);
  /* USER CODE BEGIN TIM3_IRQn 1 */
  if (led_index >= 400) {
          led_index = 0;
```

```
  }

  int x = index_to_coords[led_index][1];
  int y = index_to_coords[led_index][0];

  Charlie_SetLED(x, y, display[x][y]);

  led_index++;
  /* USER CODE END TIM3_IRQn 1 */
}
```

Code 5: Display refreshing

We will note the usage of a pre-computed index-to-coordinates map to avoid unnecessary computation during the refresh.

I then proceeded to make a simple implementation of the Game of Life using the standard rules. We won't detail it here as it's not relevant.

4.1.6 Review

This prototype effectively fills most of the functional parts.

Let's see first some pics of the whole display lit, and half lit, under a desktop lamp:

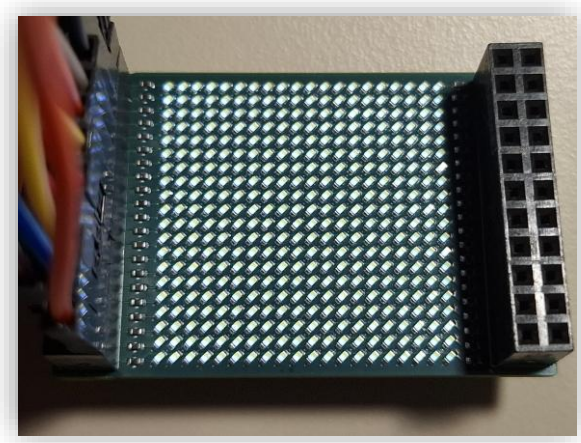


Figure 6: Prototype N°1 – Fully lit display

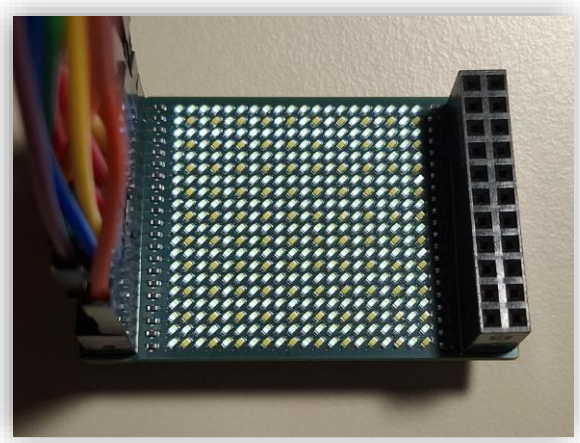


Figure 7: Prototype N°1 – Half-lit display

Th display is flicker-free to the human eye at our refresh rate but is not against a camera. Below are two videos comparing how it looks from a 30 and 60 FPS recording (clickable videos).

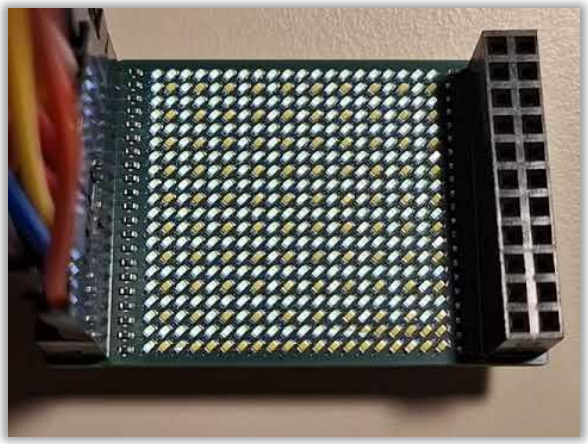


Figure 8: Prototype N°1 – 30 FPS recording (video)

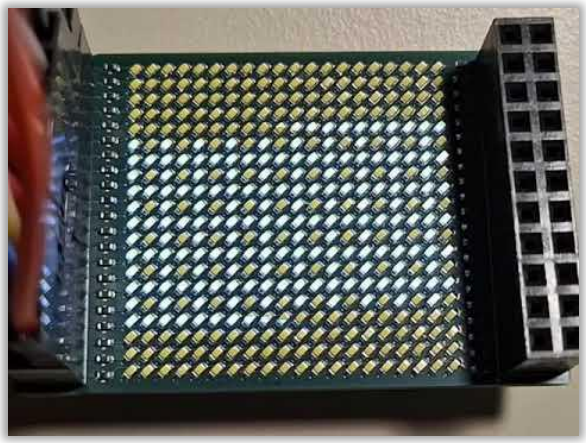


Figure 9: Prototype N°1 – 60 FPS recording (video)

And finally, what we've been all waiting for… A real demonstration! (Again, clickable video)

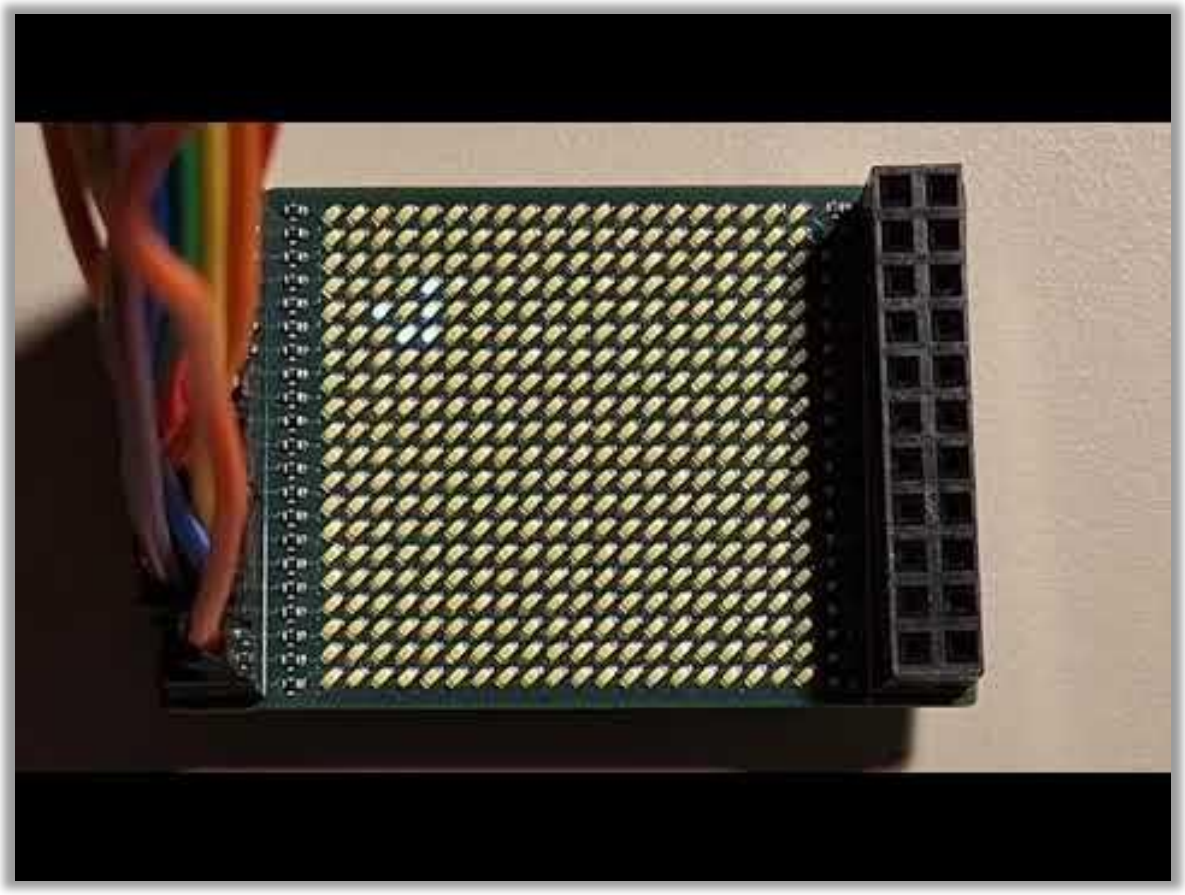


Figure 10: Prototype N°1 – Glider test (video)

And nope… I didn't alter the speed of the video… It's real speed…

Well, we can conclude from this first prototype that it "works" but the screen is rather dim, hardly seeable against a desktop lamp and not at all in sunlight. I overlooked the fact that LEDs were going to be one for a brief period and calculated the resistors for steady operation.

And the simulation, despite heavy optimizations, is astonishingly slow, mainly because I overestimated the capacities of our poor STM32F030R8.

4.1.7 Improvement and changes needed

Two big improvements are needed in the next revision:

1. **Brighter display**: this can be done by lowering the series resistors used on GPIOs, making use of their full 20 mA current source/sink capabilities.
2. **More raw processing power**: we're going to change our MCU for a more powerful one. The new one will be a STM32G431CBU6, while being twice the price of our first contestant, it has a CoreMark score 5 times greater than the F030R8 (106 v/s 569), and a clock 3.5 times faster, going from 48 to 170MHz.

4.2 Prototype N°2

4.2.1 Goal

The proof of concept being successful, this prototype will aim to be the final product by incorporating the needed changes as well as power circuitry and controls.

4.2.2 Design & diagrams

For this design, we first have the powering circuitry. It consists of a USB-C connector with 5.1 kΩ resistors on CC pins for C-to-C and a (very) small TVS diode to protect against voltage spikes originating from ESD. There is then a 250 mA LDO voltage regulator (XC6206P332MR-G) dropping down the voltage from 5 V to 3.3 V, with 10 µF ceramic capacitors (CL05A106MQ5NUN) on each side.

According to Samsung's data, we're seeing a DC bias of -76 % at 5 V (2.4 µF real capacitance) and -62 % at 3.3 V (3.8 µF real capacitance). This should be enough for us, LDO regulators often requiring a single microfarad and IC decoupling not needing much more.

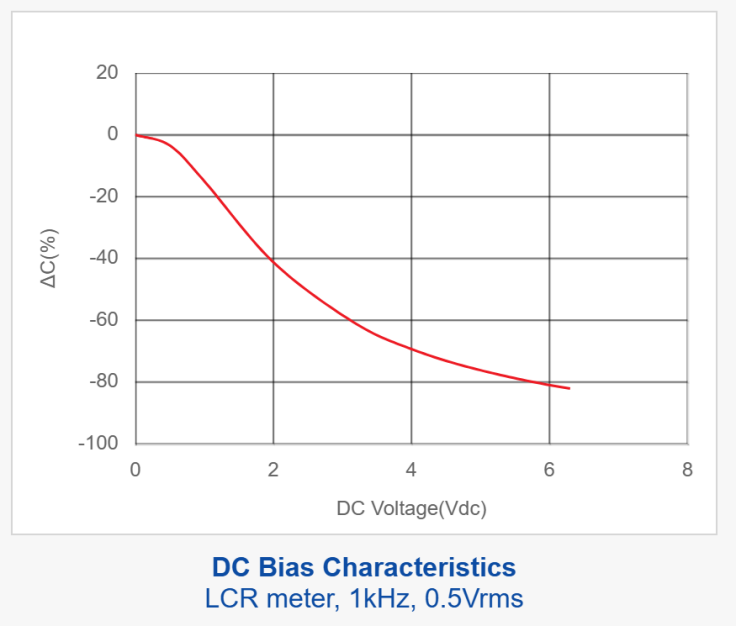


Figure 11: DC bias characteristics of the selected 10 µF ceramic capacitor (Samsung Electro-Mechanics, s.d.)

We do also have a 100 µF tantalum capacitor for bulk capacitance.

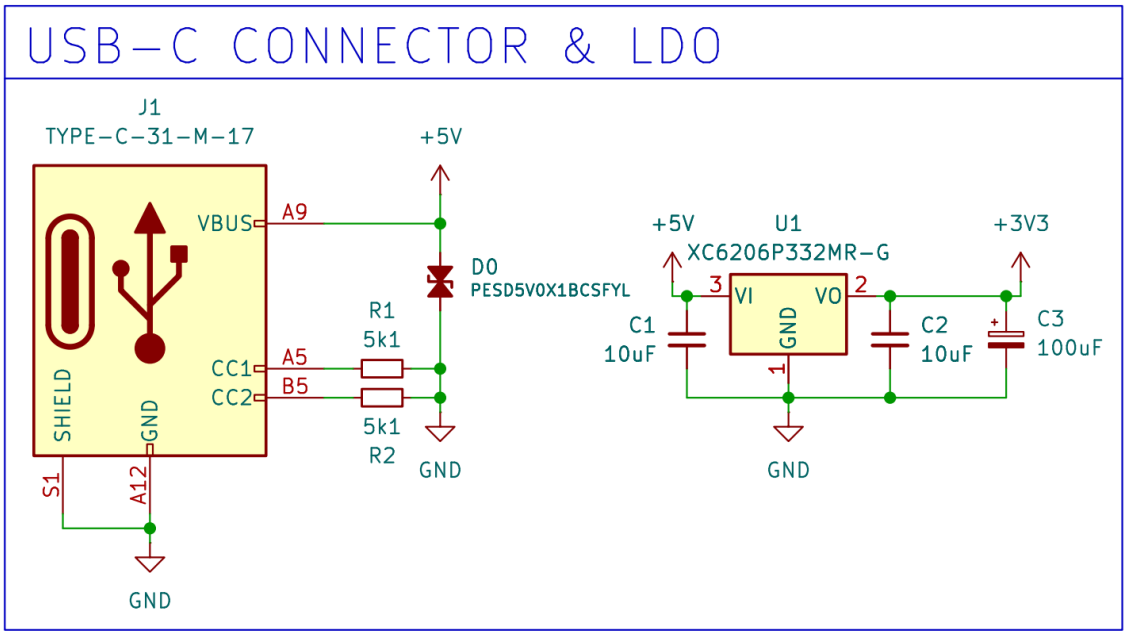


Figure 12: Prototype N°2 – Power circuitry

And to finish it we have our microcontroller implementation. It consists of decoupling capacitors, the same used for the voltage regulator, and a low-pass filter with $f_c = 503.3\ kHz$ for VDDA. We also have four programming pads for SWD (VCC, GND, SWDIO and SWCLK) to use with a ST-Link. Two buttons are also present in a pull-down configuration. Since they are not used anywhere else than in the microcontroller, we'll use the pull-ups resistor from the latter.

Finally, we have the star of the show, our STM32G431! It's powered by an external 16 MHz crystal. The crystal load capacitance is calculated using the following formula: $C_{load} = (C_L - C_{stray}) \times 2$ with C_{load} the capacitance required for each capacitor in pF, C_L the load capacitance given in the crystal's datasheet in pF and C_{stray} the parasitic capacitance from the board itself, usually estimated to be around 2 to 5 pF. With $C_L = 9\ pF$[1] and $C_{stray} = 2\ pF$, we get: $C_{load} = (9 - 2) \times 2 = 14\ pF$ per capacitor. We will choose C0G ceramic capacitors to ensure stable capacitance across a wide temperature range. 14 pF being not available is the basic parts list from JLCPCB, 15 pF was selected instead, for non-critical applications, this shouldn't be a problem.

It's also good to note that the pins distribution may seem random but was adapted to facilitate routing.

[1] Datasheet specifies 10 pF, while the LCSC page specifies 9 pF. Using 10pF gives $C_{load} = 16\ pF$, hence the selection of the 15 pF capacitors.

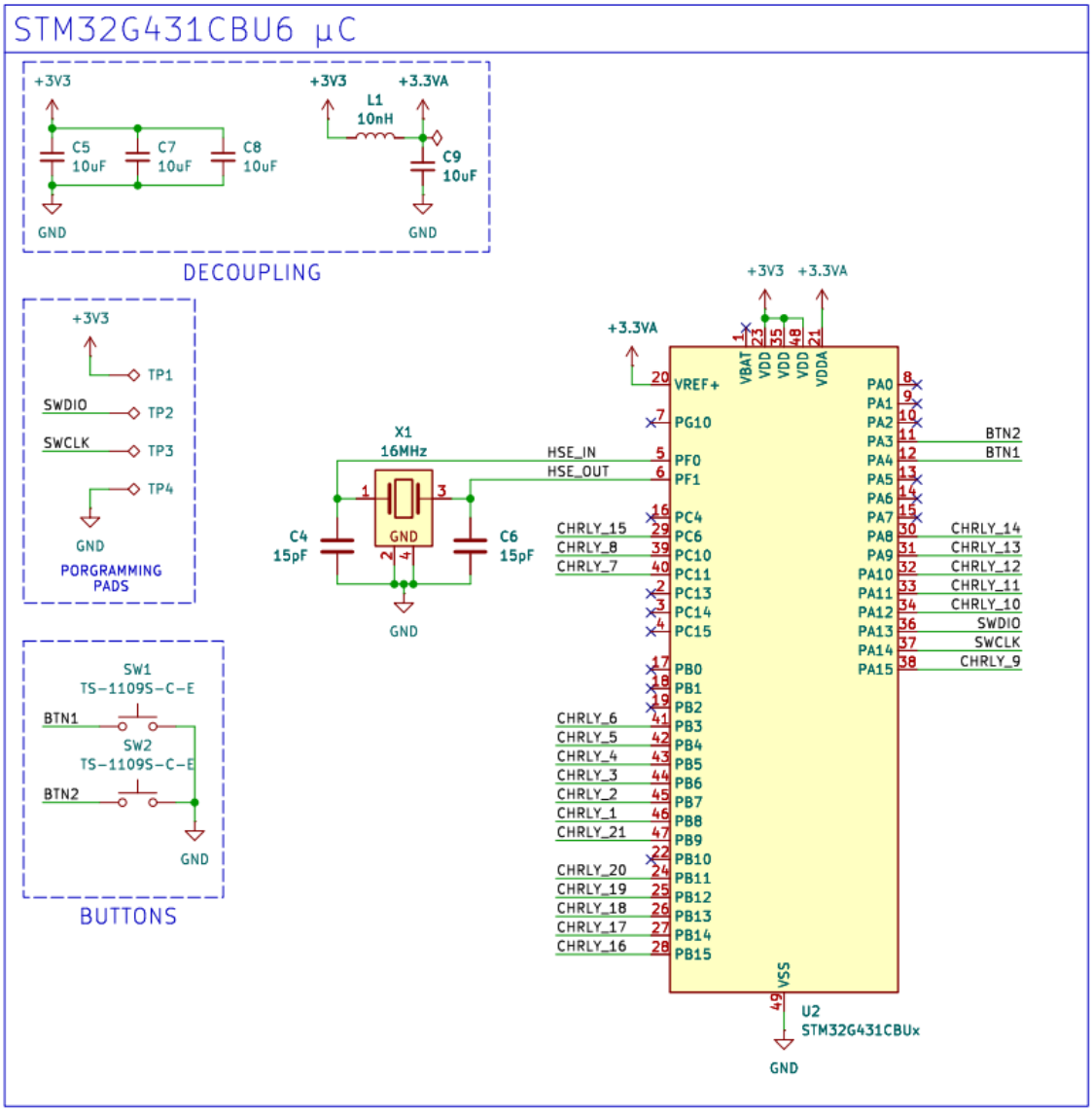


Figure 13: Prototype N°2 – STM32G431 and its peripherals

We of course have the LED matrix, but it's the same as Figure 2: Prototype N°1 – Charlieplexed LED matrix so we won't detail it right here. We'll just note the decrease of the series resistors from 11 and 22 Ω to just 1 Ω. This will allow more current to reach the LEDs and make the display a bit brighter, while keeping our current sources, the GPIOs, safe from any short-lived current spike.

4.2.3 Assembly notes

Please note that 5 boards were assembled, all numbers are for an assembly of 5 boards.

I am clearly not assembling this again by hand, so we'll make use of JLCPCB's assembly services again. The trick to saving money on assembly is to use their basic parts library, otherwise you get a $3 fee on every part not in this library, no matter the number of boards assembled, nor the number of this part used. So, the selection of parts was heavily based on this list.

We only have five parts who are not on this list:

- The STM32G431CBU6

- The TVS diode (PESD5V0X1BCSFYL)
- The LEDs (XL-1005UWC)
- The USB-C port (TYPE-C-31-M-17)
- The buttons (TS-1109S-C-E)

That's a whopping $15 in fees, shared between our 5 boards that's a meagre $3 per piece.

The rest of parts were either bought from their stock or from other users selling their idle parts, this trick also allowed us to save a few bucks on parts.

The last trick I had up my sleeve was obviously to make use of economic assembly and sadly this is where we had to sacrifice the 4L PCB for a 2L. Economical assembly is available only for a selected set of solder masks, surface finish and number of layers. And since black solder mask was absolutely needed for the esthetics of our product, we had to fall back on 2L since it's only with 2L that black solder mask is available for economic assembly.

All of this takes up to the final price. For five 2-layer, lead-free HASL, black solder mask boards, we spent €3.47 and for economic assembly, including the setup fees, feeder fees and parts cost (€25.37), we spent €40.72. LEDs alone account for more than 50 % of the total parts costs.

In the end we spent €10.61/board, including VAT (20%), excluding shipping. Find below pics of the gorgeous, assembled boards:

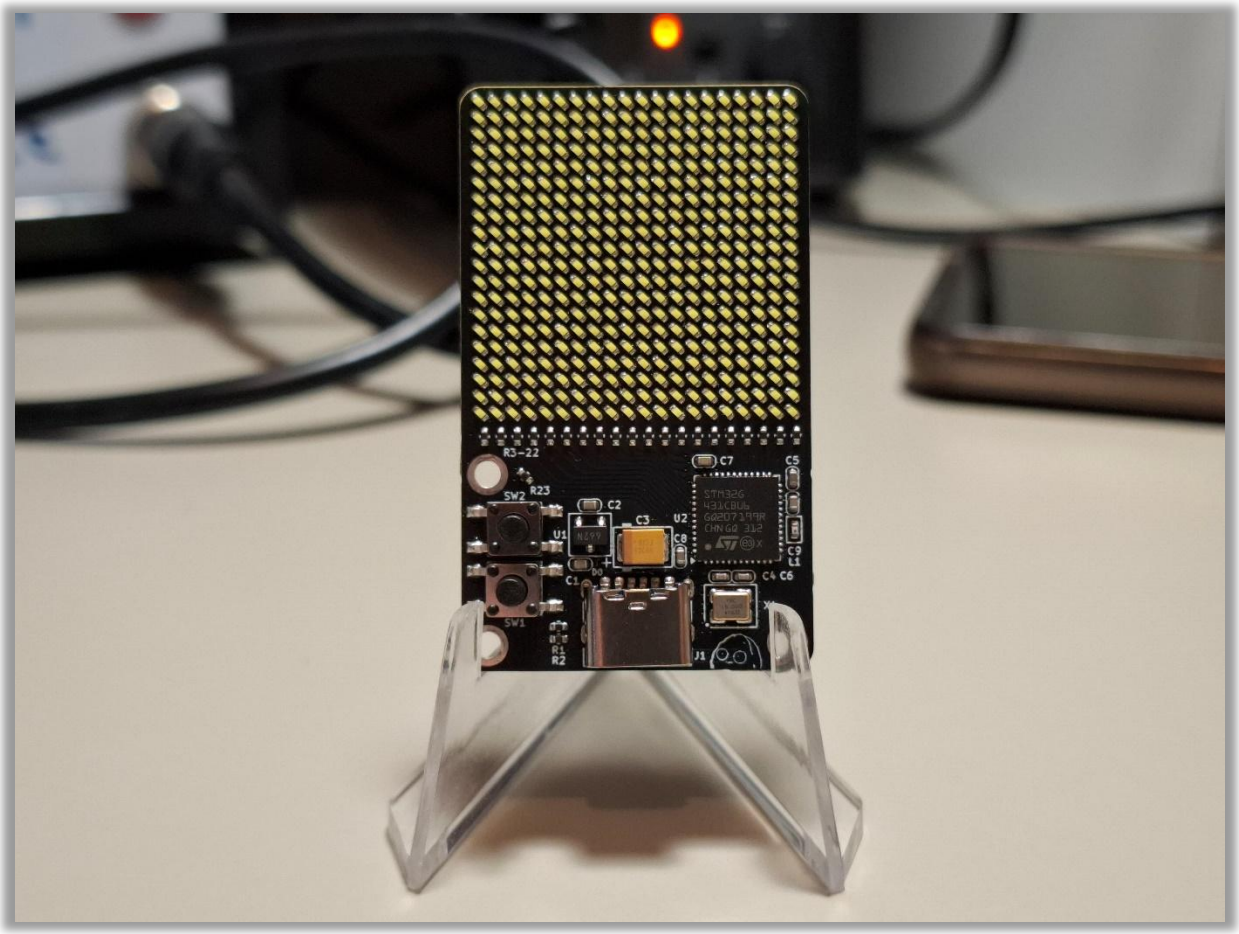


Figure 14: Prototype N°2 – Front view of the assembled board

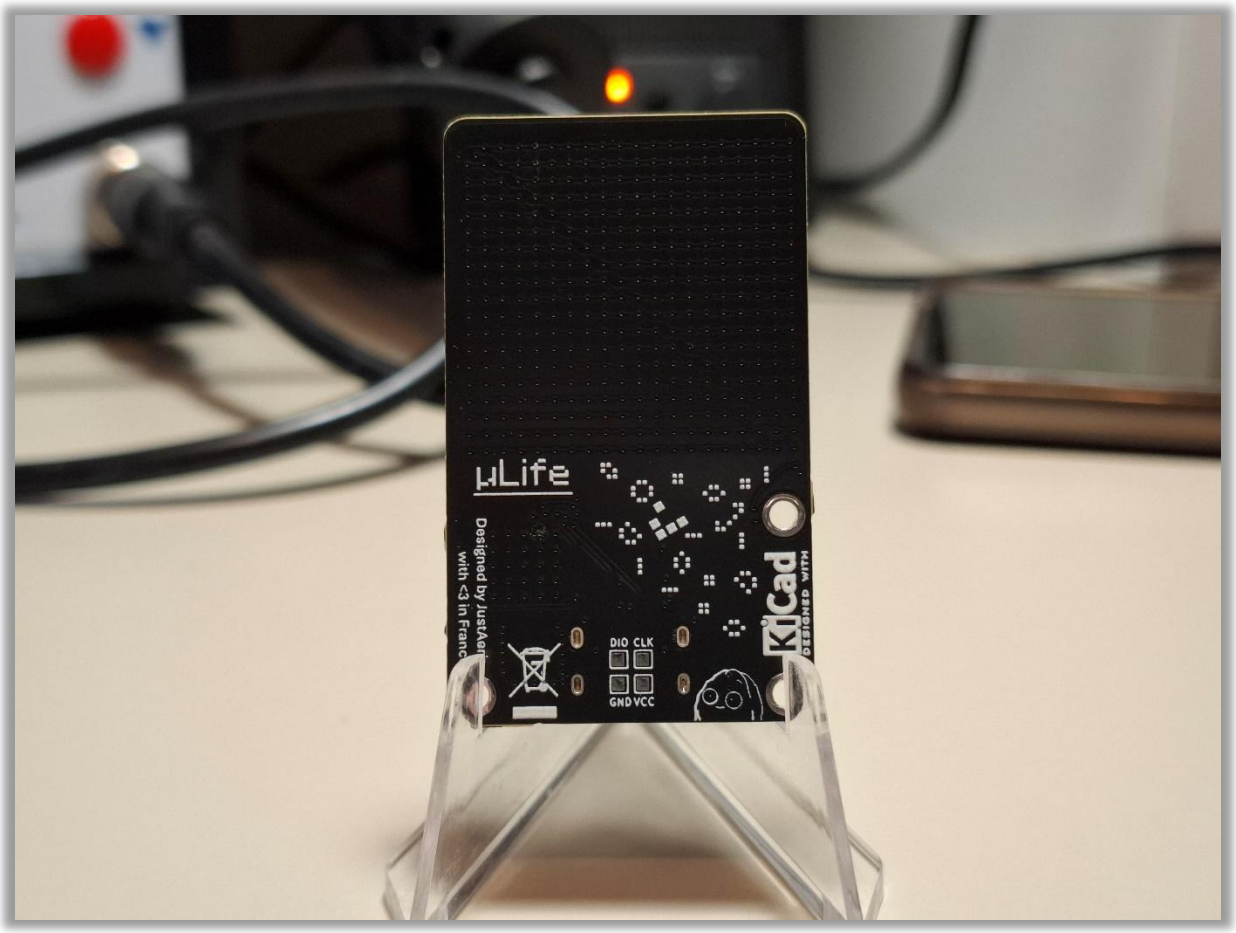


Figure 15: Protoype N°2 – Back view of the assembled board

4.2.4 Testing

Testing this board was very simple: solder four wires and see if the µC is alive. Spoiler: it is. The second step was to upload some random testing code and indeed this was a great success, see below the clickable video of the demo built the same day:

Figure 16: Prototype N°2 – Video of the first demo

Aside from that, testing was done, and it was finally time to build the firmware.

4.2.5 Software implementation

Rather than describing the code in a boring old-fashioned way block per block. I propose instead this time to take a global view on the project using diagrams. This will help to see the firmware on a larger scale.

But before, two important notes:

- The display refresh rate has been **increased to 120 Hz**. The G431 has enough computing power to keep up and it allows for nicer recording.
- The random grids are **generated using the True-RNG** module from the G431, guaranteeing unique grids each time you randomize the game. There are 2^{400} grids possible (that's a large number), meaning you'll only get grids no one had and will have in the whole universe[2]!

Let's start with the system layers:

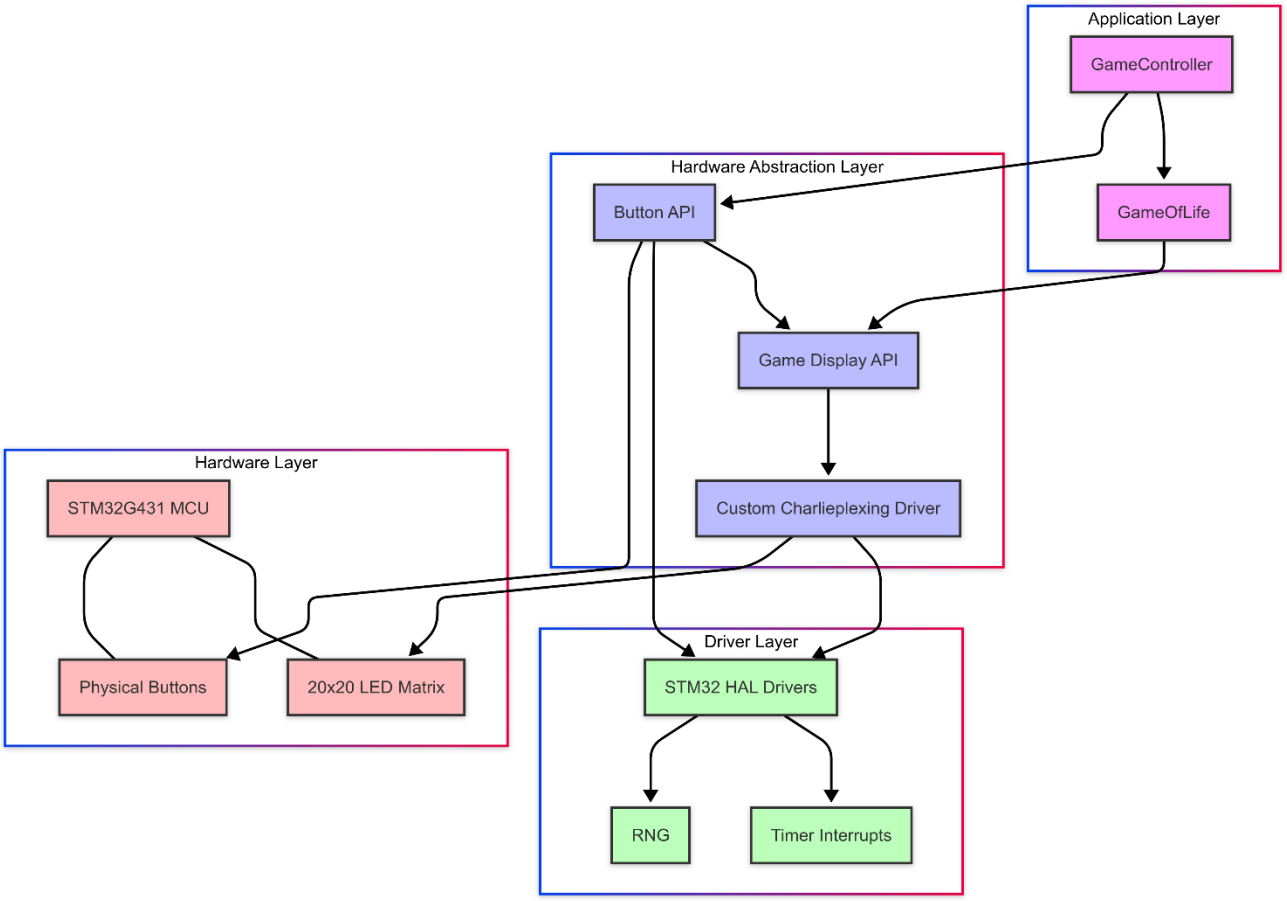


Figure 17: Prototype N°2 – System layered architecture

This architecture helps keep the system maintainable by separating application logic, hardware abstraction and drivers.

[2] Let's consider the lifetime cycle of the Universe from the Big Bang to its thermal death (approx. 10^{100} years). If we generated one random grid every second, we'd need 3 160 000 000 000 universe cycles to see every possible grid.

On a more coding side we can now take a look at the class diagram:

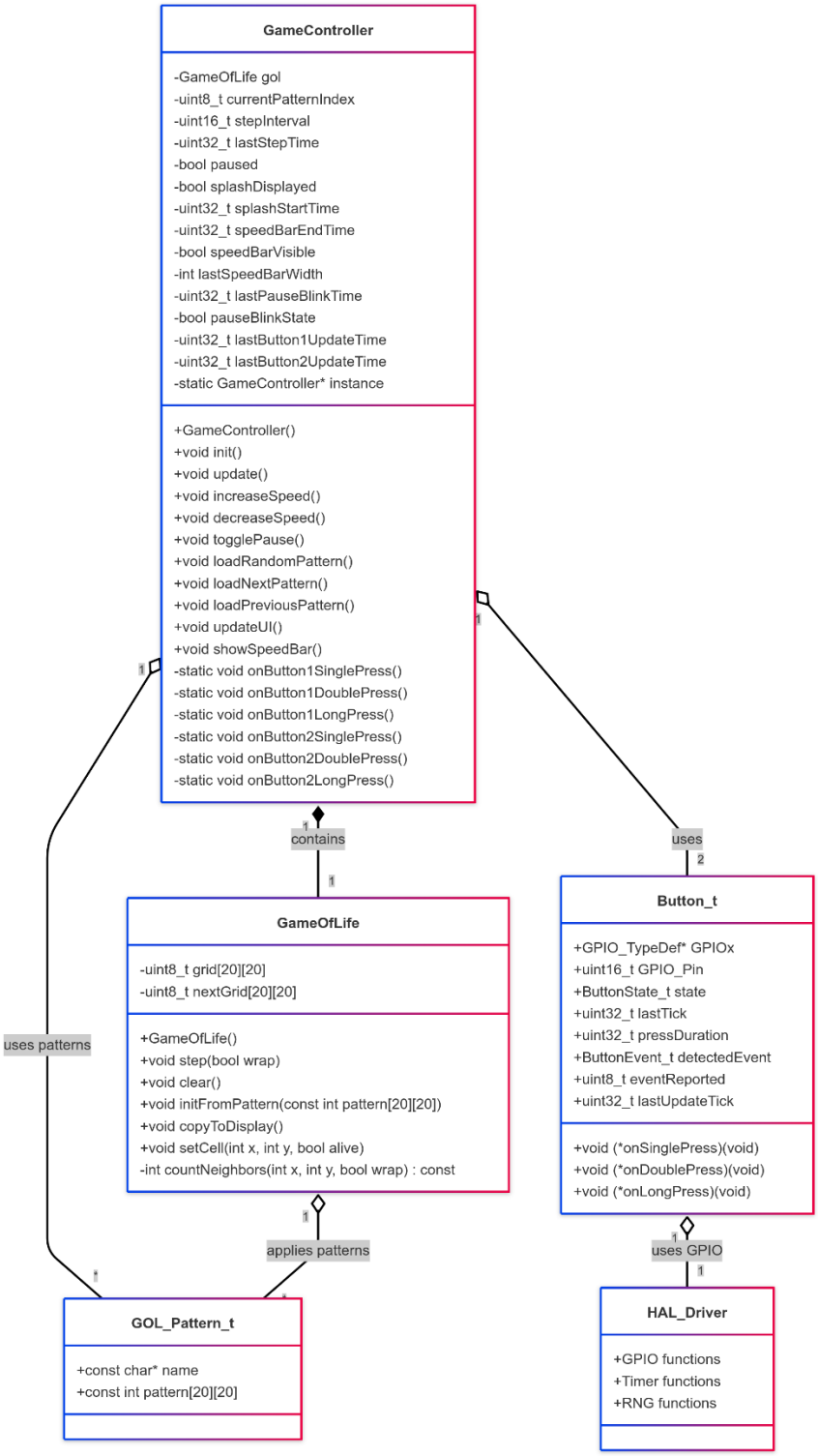


Figure 18: Prototype N°2 – UML Class diagram

We can observe again the separation. We have on one side the game engine managing the cellular automaton rules and on the other side the game controller coordinating the entire system with the help of the button management system, responsible for detecting simple, double and long presses. We also have the preloaded patterns encoded in a separate file.

Finally, one more interesting piece of information, the sequence diagram. It describes the series of events happening from the start of the game and until it ends.

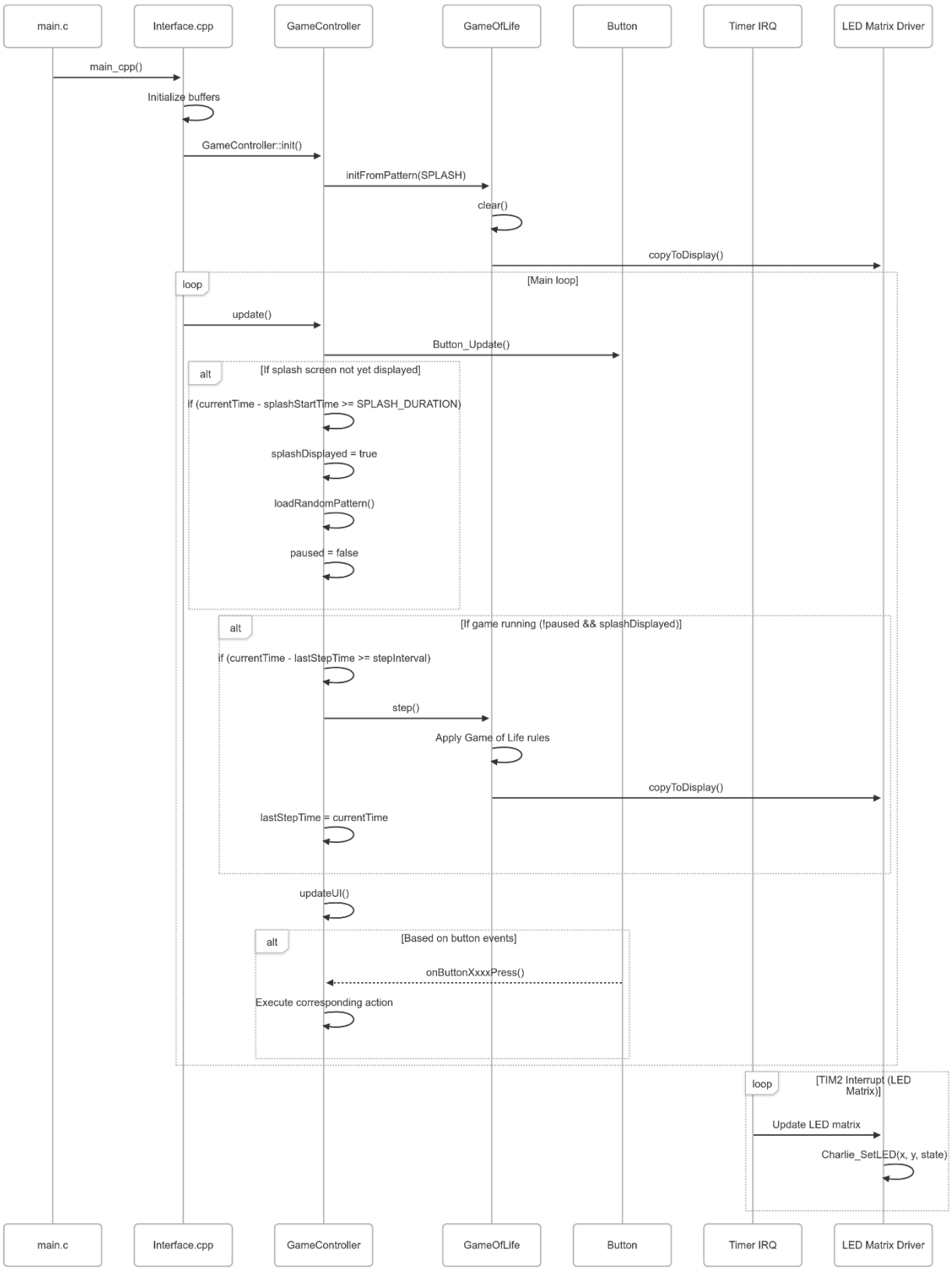


Figure 19: Prototype N°2 – Sequence diagram

4.2.6 Review

I think that this final prototype is rather successful. It fills all constraints and requirements while featuring unique features like the uniqueness of generated grids. It works greats, controls are rather easy to use when you know them.

The only two constraints we failed to comply with are the size and screen brightness ones. First, it was not possible to fit all the circuitry in 3×4 cm on a single side (which is a requirement), that's why a size of 3×5 cm was adopted instead, keeping the product rather small. Second, the display still struggles to be bright enough. Under a strong light lit LEDs become indistinguishable from the others (the display is still usable in normal conditions), sadly we can't do much about it without resorting to methods that would greatly increase the space required. At least the display is gorgeous in the dark, with lit pixels projecting a slight glow on neighboring unlit pixels.

This board will be considered final.

4.2.7 Improvements and changes needed

No prototypes will follow this one.

5 3D modeling and enclosure design

5.1 Objectives

The objective of this enclosure is to protect the product from external damage, such as scratches and impacts. Liquids and dust are out of this scope. The cover shall not apply destructive operations on the circuit, such as glue or other permanent modification, thus making the cover entirely separate and replaceable. The cover should also protect the user from toxic materials such as lead solder, if used in assembly. The cover shall also be transparent to keep the display visible.

5.2 Initial concept

The initial concept was to reproduce something similar to the case used for the SMD challenge, as in, a cover flush with the sides, like two plates covering top and bottom. However, this came with a major failure point in our situation: the sides. On the SMD challenge, parts were far enough from the sides to allow for walls or even slanted walls, such things are not possible here due to the row of LEDs we have on the edge of the board.

Because of this we'll settle on a (almost) regular case design with external walls. We'll just make the top of the USB-C connector flush with the case, since it's the same height as the body of the buttons, also in the goal of keeping the product thin. We'll also need a cutout for USB-C cables.

5.3 Design process

The enclosure consists of two pieces joined by screws and nuts. We'll define the top as the side with parts and bottom the other side.

The top case consists of a 1.6 mm thick plate with 1.5 mm thick walls on the side and a 0.5 mm clearance between board and case on each four sides. A cutout was added to make space for the USB connector, both on top for the part itself and on the side to allow space for cables. Another cutout for the buttons was made. It will allow the body of the buttons to sit flush with the top as to only have the plunger stick out. Another part of the top plate was extruded, this time on the inside, to make space for the tantalum capacitor, taller than the rest it needs extra space. And lastly, we obviously have holes for the M2 screws (EDDM-M2-L6 on JLCMC), 2.4 mm hole diameter with 4.2 mm head diameter. This accounts for a total height of 4.6 mm.

Here are views of the top cover:

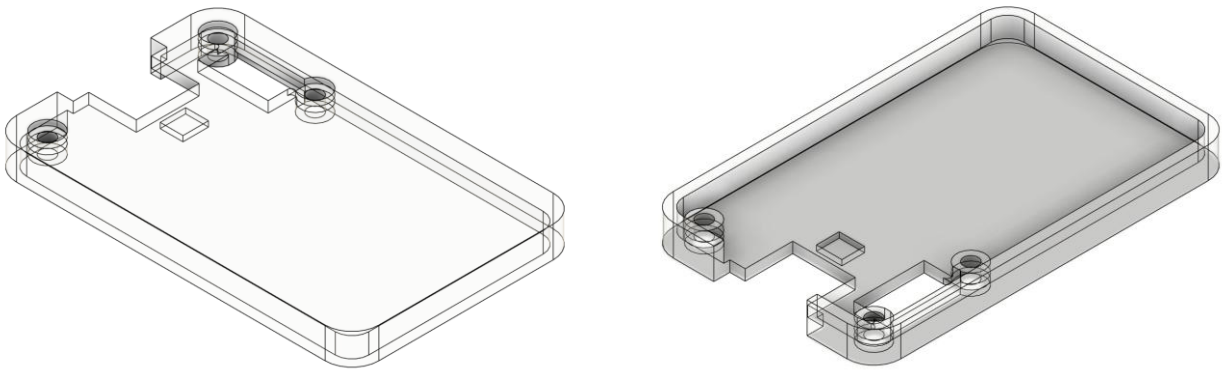


Figure 20: Front and back views of the top cover

The bottom cover is very simple. It's just a simple plate 2.2 mm thick with a side cutout of USB cables and holes for screws and nuts (EMLA-S1W-B1L1-M2 from JLCMC).

Here are views of the bottom cover:

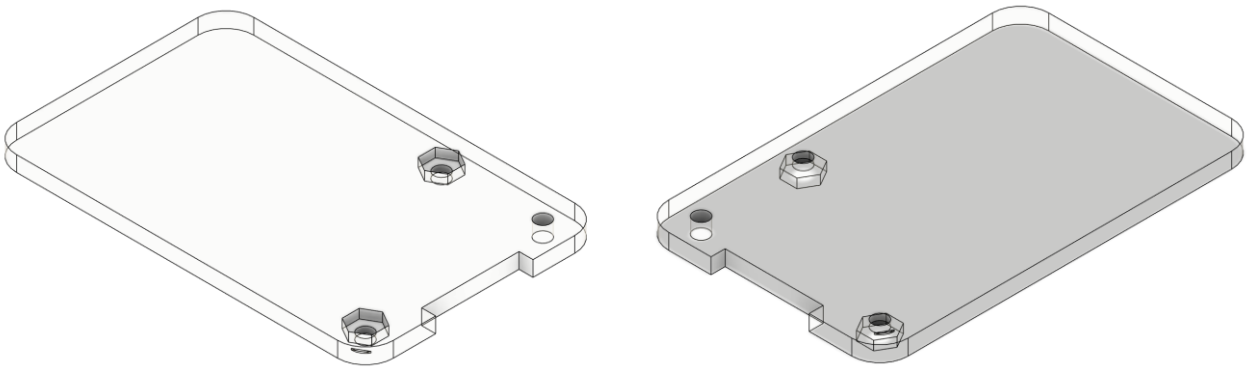


Figure 21: Front and back view of the bottom plate

This brings us to final dimensions of 34 × 54 × 6.85 mm, keeping the final product rather thin.

Here are views of the assembled models:

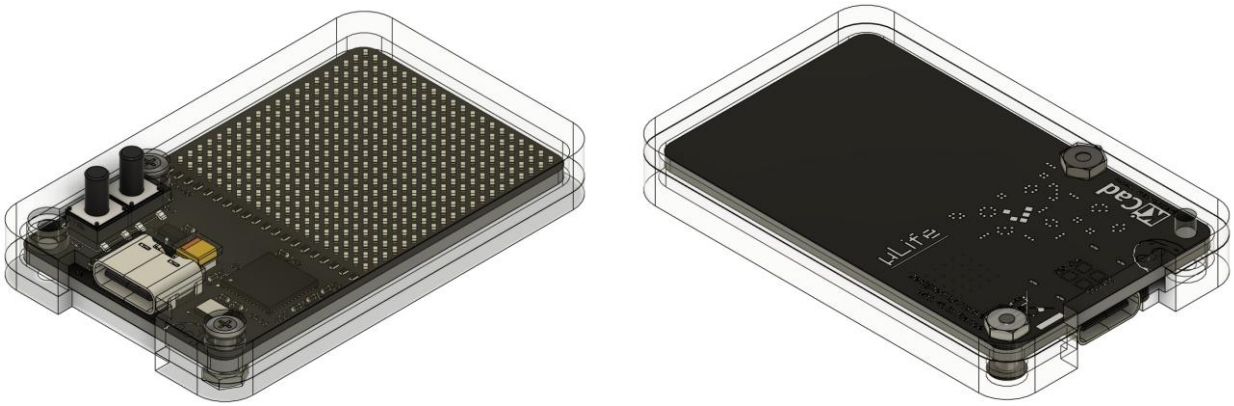


Figure 22: Full 3D view of the product

5.4 Material selection

For this project, as I'm using JLC3DP services to get my parts printed, I'll make use of their 8001 resin which has (citing JLC3DP) "excellent strength and toughness, high precision, and good dimensional stability" and aside from that, it is obviously transparent. I already ordered pieces using this material before and it should be strong enough to achieve what I want.

5.5 3D printing notes

5.5.1 Constraints

For this assembly, all constraints have been respected, they are as follow for SLA (constraints that do not apply to our design will be omitted):

- Min. design size (mm): 5x5x5 OR 10x2x2
- Wall thickness: 0.8 mm
- Model clearance: 0.2 mm
- Holes design: for an aperture of 2.0 mm, the height must be at least 2.0mm

- Printing tolerances:
  - Model tolerances: ± 0.2 mm (within 100mm), ± 0.3 % (above 100mm)
  - Hole tolerances: ± 0.3 mm

5.5.2 Surface finish

For this print, we need a transparent finish, this is why oil spraying finish was selected.

5.6 Assembly fit & review

The case is almost perfect! Everything fits nicely, the cutout of the USB-C cable is large enough, the cutout for the tantalum is also pinpoint sized. The screws fit in perfectly, being flush with the surface, same for the nuts.

The first problem is caused by the screws' placement on the original board. Due to the screws being only at the bottom of the board, we have no downward pressure on the top of the case and due to the flexible nature of the resin used, this leaves a small (around 0.5 mm I'd say) gap between the two parts on the top left corner only.

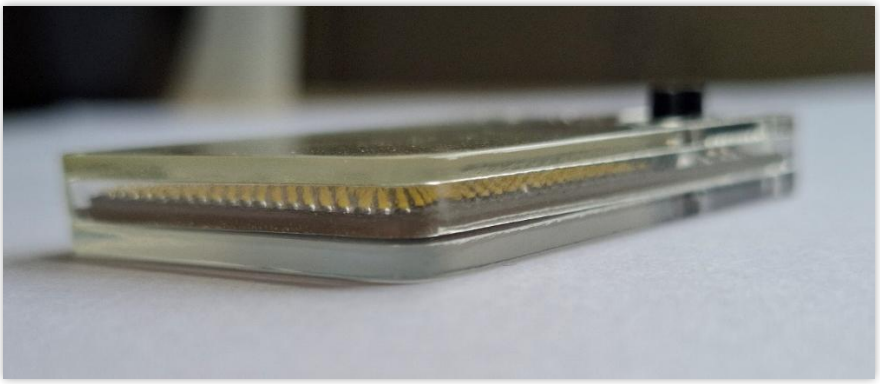


Figure 23: The gap between the two cover parts

There is another major twist, being that the resin is absolutely not scratch proof. For something that is supposed to hang with your keys, that's problematic.

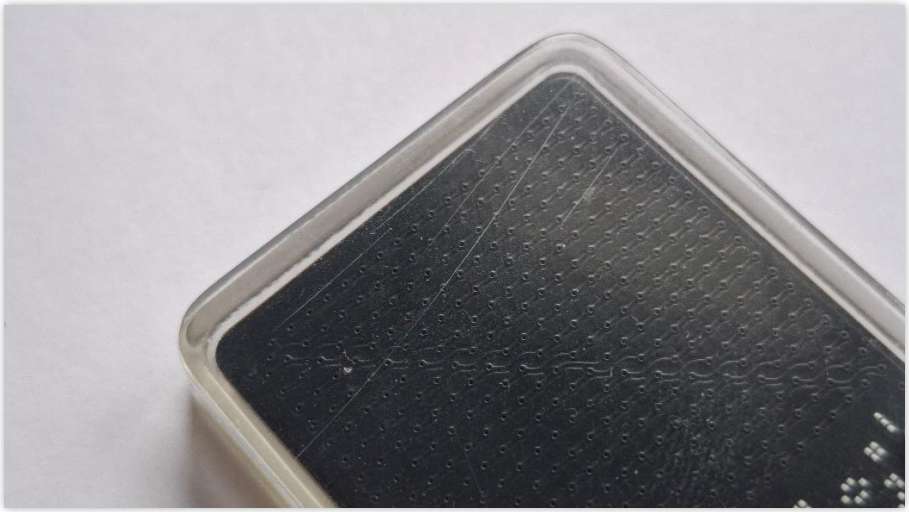


Figure 24: Scratches on the back of the case

To solve this problem, we're going to try 2K clear coat, a car varnish. It's supposed to be very thin and to form a very hard protective layer that's supposed to be scratch resistant. It'll be tested in the future.

5.7 Improvements for future cases

Sadly, we have nothing to do to improve it as it depends on the holes' placements and the material used, which we can't easily change it.

6 Final version

6.1 Final design

The final design is essentially prototype N°2 with its 3D-printed enclosure. We have a 3×5 cm PCB featuring a 20×20 LED matrix controlled by our STM32G431CBU6. The board includes USB-C power input with the XC6206P332MR-G LDO regulator, two TS-1109S-C-E tactile buttons for user interaction, and the transparent resin enclosure from JLC3DP. The charlieplexing technique lets us control all 400 LEDs using only 21 GPIO pins, which I think is pretty neat for such a compact design.

6.2 What it does

The µLife keychain runs Conway's Game of Life on its LED matrix display. Users can cycle through pre-programmed patterns like gliders, oscillators, and other classic configurations using the two buttons. The device can also generate random "soup" patterns using STM32's True-RNG module, meaning you'll get grids no one had and will have in the whole universe (as I calculated earlier). Speed control is available, and the 120 Hz refresh rate provides smooth visual updates without flicker. Just plug it into USB-C and it starts running immediately.

6.3 Performance evaluations

The STM32G431 performs way better than the original F030R8. With its 170 MHz clock and 569 CoreMark score, it handles the 120 Hz display refresh while running Game of Life calculations without breaking a sweat. The 120 Hz refresh rate also gives us almost flicker-free recording, which is a nice bonus for making demo videos. Current consumption stays well under our 250 mA requirement, typically around 150 mA during normal operation. The display brightness is definitely improved from prototype 1, though it still struggles against strong sunlight. At least it looks gorgeous in the dark with that nice glow effect between pixels.

6.4 Challenges and solutions

The biggest challenge was definitely getting enough brightness from the charlieplexed matrix. We fixed this by dropping the series resistors from 220Ω and 100Ω down to just 1Ω, pushing the LEDs to their limits while keeping the GPIOs safe. Another possibility to improve readability would be adding a transparent black plastic sheet over the display (like the tinted film they put on car headlights) - this might increase contrast further and make individual pixels more distinct, though I haven't tested this yet. Routing 400 LEDs on a 2-layer PCB was... interesting. Thank goodness for JLCPCB's basic parts library strategy, which saved us a fortune in assembly fees. The enclosure design had to work around those edge LEDs, so we couldn't do the flush design I originally wanted. Manufacturing came out to €10.61 per board, which I think is pretty reasonable considering what we're getting.

7 Conclusion

7.1 Reflection

This project turned out way better than I expected when I first got the idea watching bitluni's videos. The charlieplexing worked beautifully, and seeing Conway's Game of Life running on a keychain still gives me that "wow, I actually built this" feeling. Sure, we didn't hit the original 3×4 cm target (ended up at 3×5 cm), and the brightness could be better, but honestly? It does exactly what I wanted it to do. The software architecture with proper separation turned out to be crucial - I'm glad I took the time to do it right instead of just hacking something together.

7.2 Future of the project

I've already decided the next version will be a complete redesign using an OLED screen for that crisp HD experience. No more brightness issues, no more charlieplexing headaches, just pure pixel perfection. The OLED will also allow for a much larger grid size, giving Conway's Game of Life more room to develop complex patterns. The current 20×20 grid is quite limiting for some of the more interesting long-term behaviors.

The OLED version will probably let me shrink the whole thing down significantly too, maybe even hit that original 3×4 cm target. I'll also need to address the enclosure material - the current resin scratches way too easily for something that's supposed to hang with your keys. Either a different material or that 2K clear coat I mentioned should solve the scratch resistance problem.

The current version serves as an excellent foundation, but an HD OLED version with a larger grid and proper scratch resistance will be the real showstopper for the keychain.

8 Appendices

8.1 Datasheets

- NUCLEO-F030R8

https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

8.2 Bill of materials

Designator	Footprint	Qty.	Value	Comments	LCSC Part #
C1, C2, C5, C7, C8, C9	0402	6	10uF	6.3 V	C15525
C3	3528 (21 mm)	1	100uF	6.3 V	C16133
C4, C6	0402	2	15pF		C1548
D0	DSN0603-2	1	PESD5V0X1BCSFYL		C2443412
D1-400	0402	400	XL-1005UWC		C20613596
J1	HRO_TYPE-C-31-M-17	1	TYPE-C-31-M-17		C283540
L1	0402	1	10nH	300 mA	C27147
R1, R2	0402	2	5k1		C25905
R3-23	0402	21	1R		C25086
SW1, SW2	SW-SMD_4P-L4.5-W4.5-P3.00	2	TS-1109S-C-E		C561507
U1	SOT-23	1	XC6206P332MR-G		C5446
U2	QFN-48-7x7mm	1	STM32G431CBUx		C529356
X1	3225-4Pin	1	16MHz		C13738

8.3 Sources and references

bitluni. (2024, February 24). *From blink to DIY Mini Game - Charlieplexing explained*. Retrieved from YouTube: https://www.youtube.com/watch?v=OW_Sk_dbQm8

Deegan, P. (n.d.). *psychogenic/kicad-skip: kicad s-expression schematic/layout file manipulation.* Retrieved from GitHub: https://github.com/psychogenic/kicad-skip/blob/main/src/skip/examples/charlieplex.py

JLCPCB. (2024, december 26). *3D Printing Design Guideline*. Retrieved from JLC3DP: https://jlc3dp.com/help/article/3D-Printing-Design-Guideline

JLCPCB. (n.d.). *Hex nut Standard/Thin/Thickened Coarse/Fine thread*. Retrieved may 4, 2025, from JLCMC: https://jlcmc.com/product/s/E04/EMLA/FA-%E7%B4%A7%E5%9B%BA%E9%9B%B6%E4%BB%B6-%E8%9E%BA%E6%AF%8D

JLCPCB. (n.d.). *Phillips Ultra Thin Head Screw*. Retrieved may 4, 2025, from JLCMC: https://jlcmc.com/product/s/E02/EDDM/FA-%E7%B4%A7%E5%9B%BA%E9%9B%B6%E4%BB%B6-%E8%9E%BA%E9%92%89

PETERSEN, L. (2021, October 21). *How to generate a one second interrupt using an STM32 Timer.* Retrieved from ST Community: https://community.st.com/t5/stm32-mcus/how-to-generate-a-one-second-interrupt-using-an-stm32-timer/ta-p/49858

Samsung Electro-Mechanics. (n.d.). *CL05A106MQ5NUN | MLCC | Component Library*. Retrieved may 4, 2025, from https://weblib.samsungsem.com/mlcc/mlcc-ec-data-sheet.do?partNumber=CL05A106MQ5NUN

Wikipedia. (n.d.). *Charlieplexing*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Charlieplexing