ChatGPT

Programmable LED Pattern Language Architecture (ESP32 + Mobile App)

Introduction

Designing a programmable LED pattern language for ESP32-based controllers requires balancing flexibility, efficiency, and ease of use. This system will enable dynamic, modular light patterns that can be created and shared in real-time. The architecture consists of a **high-level pattern definition language** (using modular "cells"), a lightweight **runtime interpreter** on the ESP32, and a **mobile app** (Cordova-based) that communicates over BLE. Patterns are designed to be compact (only a few KB) and support live parameter tweaking (colors, speeds, etc.) from the app. Unlike code-centric approaches (e.g. Pixelblaze's JavaScript-like language [1] [2]), this design adopts a higher-level, building-block paradigm. Each pattern is composed of reusable visual effect modules ("cells") instead of low-level math code. This makes pattern creation more accessible and allows patterns to be easily shared, evolved, and synchronized across multiple devices.

System Overview

- **ESP32 LED Controller**: Runs the pattern interpreter/VM and drives the LED strip or matrix. It stores a library of patterns (in a compact format) and executes the currently selected pattern, updating LEDs frame by frame.
- **Cordova Mobile App**: A cross-platform app (Android/iOS) that provides a user interface to select patterns, adjust parameters in real time, and manage pattern sharing. It communicates with ESP32 devices via Bluetooth Low Energy (BLE) to send pattern definitions, parameter updates, and to query/transfer patterns.
- **BLE Communication**: A custom BLE GATT service connects the app to the ESP32. This provides characteristics or commands for selecting a pattern, sending a new pattern definition (few kilobytes), and updating parameters (booleans, sliders, color picks, etc.) on the fly. The BLE link enables real-time control with minimal latency.
- **Pattern Definition Language**: A domain-specific format (JSON or bytecode) describing a pattern as a network of "cells" (effect modules) plus their parameters. This definition is compact and efficient to parse on the device. Patterns are inherently 1D (a linear sequence of LED effects), but can be *mapped* onto 2D LED layouts via configurable mapping modes (e.g. horizontal, vertical, radial projection).
- **Server/Cloud (optional)**: Patterns can be synced to a central server via the mobile app when internet is available. This allows backing up user-created patterns and sharing popular patterns globally. The app can also facilitate **peer-to-peer pattern sharing**: acting as a bridge to copy patterns from one ESP32 device to another, even if no direct internet is present. The system is designed to allow patterns to "spread" virally between devices (with potential **evolution** of patterns as they propagate) and later upload new variants back to the server.

Pattern Definition Format

To ensure portability and small size, patterns are defined in a compact data format. A practical choice is **JSON** (or a binary equivalent like CBOR) to represent the pattern structure, since JSON is human-readable and easily handled in a Cordova (JavaScript) environment. Each pattern definition includes:

- **Metadata**: A unique pattern ID, name, version, and perhaps author or origin. The ID (e.g. a GUID or hash) is used to identify the pattern during transfers and avoid duplicates.
- **Cell List (Effect Graph)**: An ordered list of "cells," each representing a building-block effect or transformation. Each cell entry includes a `type` (identifier of the effect module) and a set of **parameters/config** specific to that effect. For example:
- A gradient generator cell might have parameters defining its color palette or colors, and maybe a direction or orientation.
- A sine-wave modulation cell might include parameters for amplitude and frequency (which could be tied to a speed slider).
- A blending cell may reference two sub-patterns or layers and include a blending mode (add, multiply, crossfade, mask, etc.).
- **Real-Time Parameters**: A section defining external tweakable parameters (if any) that the pattern exposes. For each parameter: name, data type (e.g. boolean, float slider, color picker, list selection), default value, and range or options. These parameters can be bound to cell properties. For instance, a "speed" slider might be bound to the frequency of a wave effect cell, or a color picker bound to the base color of a gradient cell. The definitions here allow the mobile app to dynamically render appropriate UI controls for the user.

Example (JSON snippet):

```json
{
  "id": "pattern123",
  "name": "Rainbow Wipe",
  "cells": [
    { "type": "gradient", "palette": "rainbow" },
    { "type": "wipe", "direction": "left", "speedParam": "speed" },
    { "type": "sineMod", "amplitude": 0.5, "frequency": 1.0 }
  ],
  "params": [
    { "name": "speed", "type": "slider", "min": 0.1, "max": 5.0, "default":
1.0 },
    { "name": "invert", "type": "boolean", "default": false }
  ]
}
```

In this example, the pattern has three cells: a color gradient (with a "rainbow" palette), a wipe effect that uses a parameter "speed" for its movement speed, and a sine-wave modulator. Two real-time parameters are defined: a continuous speed slider and a toggle (`invert`). The JSON is compact (only a few hundred bytes) and can be further minified or converted to a binary form for BLE transfer. Using a binary format (a custom bytecode or TLV encoding) could reduce size by omitting field names; however, JSON's readability and the small absolute sizes (typically <2 KB per pattern) make it a reasonable choice. The Cordova app can easily construct or parse these JSON patterns. If needed, the app could **compile** the JSON into a concise bytecode before sending to the device – essentially translating the cell list into opcodes – to minimize on-device parsing. This compilation can be done in JavaScript, as seen in similar projects (e.g. LSSL compiles an LED shader language to bytecode in the browser) [3] [4]. The pattern format is designed to be *forward-compatible*: new cell types or parameters can be added in firmware updates, and older devices would ignore or fail gracefully if they encounter unknown types.

Modular Pattern Cells and Composition

At the core of the language is the concept of **cells** – independent modules that each perform a specific visual function. Patterns are created by layering or chaining these cells. This modular design means complex animations can be built by **composing simple building blocks**. Key cell categories include:

- **Generators**: Produce a base pattern or color output, usually based on pixel position (and possibly time). Examples: a solid color fill, a two-color gradient across the strip, a rainbow spectrum, a Perlin noise field, or a sparkles generator. Generators typically have no input from prior cells – they start the pipeline with an initial color for each pixel.
- **Modifiers/Filters**: Transform or modulate the pixel colors fed into them. They take an input (from a generator or a previous modifier) and apply an effect. Examples:
- *Sine wave modulator* – adjusts brightness or color of each pixel with a sinusoidal curve (could create a pulsing or wave motion). For instance, it might multiply the pixel's brightness by `0.5 * (1 + sin(phase))`, where phase might vary over the strip or time.
- *Color shift/hue rotate* – shifts the hue of all pixels by a certain amount (possibly animated over time).
- *Intensity/Contrast* – globally brightens or dims the pattern, or increases contrast.
- *Blur* (if supported) – a filter that averages colors with neighbors to create a smoothing effect (this one would require reading neighboring pixel values, which is more complex, so it might be an advanced cell type).
- **Masking and Layering**: Combine two streams of pixel data or selectively apply effects:
- *Mask cell* – takes two inputs (foreground and background patterns) and a mask pattern (which could be procedural). It allows the foreground to show through in some regions and the background in others (e.g. a shape or a moving wipe defines the mask). For simplicity, the mask might be implicitly generated by parameters (like a threshold, shape or gradient).
- *Blend cell* – combines two full patterns or layers into one, using a specified blend mode (additive blending, crossfade with a certain alpha, etc.). This effectively enables parallel branches in the pattern: two sub-patterns are rendered and then merged. For instance, one could blend a moving sparkles pattern on top of a steady color glow.

Composition and Execution: By default, patterns will be defined as a **linear pipeline** of cells (the output of one feeds into the next). This covers a wide range of effects with minimal overhead – essentially a *functional composition*. For more complex scenarios (like blending or masking two patterns), the pattern can include a special combinator cell that has multiple input references. In the pattern JSON, this could be represented by nested cell definitions or by referencing earlier cells. For example, a blend cell might look like:

```
{ "type": "blend", "mode": "add", "inputA": 0, "inputB": 1 }
```

which instructs the runtime to take the outputs of cell 0 and cell 1 (two separate sub-patterns defined elsewhere) and blend them. Internally, the runtime might handle this by evaluating those sub-pattern branches separately and then applying the blend. To keep the interpreter simple, one approach is to **evaluate sub-patterns sequentially** into frame buffers and then combine them. While this uses a bit more memory (multiple frame buffers), it simplifies processing of multi-input nodes. Efficient composition can be achieved by reusing buffers and only allocating what's needed for active patterns. For instance, if a mask uses a simple analytic function (like a radial gradient) rather than a full image, the mask can be computed on the fly per pixel without an extra buffer. The goal is to support layering without significant performance loss by optimizing common cases (e.g. if a mask is static or simple, it needn't be recomputed every frame).

Efficient Chaining: In the typical linear case, layering cells is very efficient. The ESP32 can iterate over LEDs and apply each cell's operation in sequence for each pixel. This avoids creating large intermediate arrays for each cell. Pseudocode for the render loop per frame might be:

```
for (int i = 0; i < numPixels; ++i) {
    Color color = {0,0,0};
    for (Cell *cell = pattern.firstCell; cell != NULL; cell = cell->next) {
        color = cell->apply(color, i, time);
    }
    frameBuffer[i] = color;
}
```

Here each `cell->apply()` is a function implementing that cell's effect. For example, a **gradient cell** might ignore the input color and output a color based on the pixel's normalized position (e.g. a smooth blend from blue to red across the strip). A **sineMod cell** might take the current color and modulate its brightness by `sin(f * i + ω * t)` (where *f* is a spatial frequency and *ω* a temporal frequency derived from a speed parameter). Because each cell is a fixed, optimized routine in C/C++, this approach is essentially like a custom **DSP pipeline** – highly efficient. The overhead per cell per pixel is just a function call and a few math operations. If a pattern has *C* cells and *N* pixels, the complexity is O(C×N). For typical values (e.g. C=5 cells, N=300 LEDs), this is very lightweight. Even at thousands of LEDs, the ESP32 (with its 240 MHz clock and FPU) can handle moderate C×N at decent frame rates [2]. For instance, Pixelblaze (ESP8266 at 160 MHz) could handle ~12,000 pixel updates per second for complex patterns [5]; an ESP32 with our optimized VM can do significantly more. We can further optimize by using **fixed-point math** or LUTs for expensive functions. (Pixelblaze uses 16.16 fixed-point for all its math to speed up arithmetic on microcontrollers [1].) On ESP32, single-precision floating point is hardware-accelerated, so we might use floats for simplicity and still achieve good performance. For periodic functions like sine or noise, we can use precomputed tables or approximations to avoid heavy computation in inner loops.

Modularity Benefits: The cell architecture means new effect modules can be added to the firmware without changing the overall system. Users (or developers) could extend the pattern language by writing a new cell in C (for the ESP32) and adding its definition to the app's palette of blocks. Because patterns are defined abstractly, a user doesn't need to write code – they can select and configure cells in the app (imagine a visual editor or a form-based interface to pick effects and set parameters). This high-level abstraction makes it easier to create complex effects ("strobe a rainbow with sparkles and a moving spotlight mask") by combining simpler pieces, rather than writing math formulas from scratch.

1D and 2D Mapping Support

Patterns are inherently created in a one-dimensional space (thinking of LEDs indexed 0…N-1 along a strip). However, many LED setups are 2D (grids, rings, etc.) or even 3D. The system supports applying any 1D pattern to higher-dimensional layouts through **mapping modes**. The idea is to provide a coordinate mapping that translates each LED's position into a 1D "position" used by the pattern. This allows reusing the same pattern on different installations with minimal changes.

LED Coordinate Mapping: Each device can be configured with a coordinate map for its LEDs (for example, an array of (x,y) coordinates for a panel or a (r,θ) for a circle). The pattern runtime can use these coordinates when computing effects. We define several mapping modes that dictate how the 1D pattern logic interprets the coordinates:

- **Horizontal Mapping (X-axis)**: The pattern's internal position is taken from the LED's x-coordinate. For a matrix, this means each column corresponds to a portion of the pattern. For example, a gradient pattern in horizontal mode will produce vertical stripes of color across the matrix (the color changes as x increases, but is constant along any column). If the matrix is 16x16 and we have a left-to-right gradient, the left edge might be blue and the right edge red, with each column a blend in between.
- **Vertical Mapping (Y-axis)**: Similarly, use the y-coordinate as the position. A vertical gradient would create horizontal color bands top-to-bottom. A vertical wipe effect would sweep from top row to bottom row, etc.
- **Radial Mapping (Center-Out)**: Interpret the pattern's position as the radial distance from a defined center point. For example, on a circular or 2D cluster of LEDs, an expanding circle pattern could be achieved by a wipe or gradient in radial mode – it would treat the LED closest to center as position 0 and the farthest as position 1. This mode is great for rings or round layouts: a wave in radial mode becomes concentric rings of light expanding outward. "Radial" in this context typically uses distance = sqrt($x^2 + y^2$) (normalized by the max radius of the layout).
- **Angular Mapping (optional)**: Another possible mode (not explicitly listed in the question, but related to radial) is using the angle (θ) around a center. For instance, an angular mapping would map 0…1 to 0–360°, causing patterns to wrap around a circle or donut shape. Combined with radial, this can address polar coordinates. (If not required, we can omit this, but it's an easy extension if needed.)

In implementation, the ESP32 device will store a list of coordinates for each pixel (likely normalized to [0,1] or in actual units like centimeters). When rendering, if a pattern is in a 2D mapping mode, the runtime calculates a **position value** for each pixel according to the mode: e.g., `pos = x_norm` for horizontal, `pos = y_norm` for vertical, `pos = distance_norm` for radial. This `pos` (0 to 1) is then used in place of the usual index fraction in the cell computations. Essentially, the cells themselves can be written to use an abstract position input – by default it's `i/N` for 1D, but if a mapping is active, the runtime supplies a different value. For example, a *gradient cell* usually does `color = palette( index / (N-1) )`. Under radial mapping, it would do `color = palette( distance_from_center / max_distance )`. We can implement this by giving each cell access to the pixel's coordinate (x,y) and a mapping mode context. Some cells might ignore mapping (e.g. a pure noise might use 2D noise if available, but if not, it could just use 1D index). Many generative cells and modulating cells can naturally leverage one-dimensional position in this abstract way. The result is that a pattern "thinks" it's drawing a linear effect, but it can appear in structured ways on a panel.

The **mobile app** can expose mapping settings when the user applies a pattern to a device. For instance, if the device is set up as a matrix, the user might choose between horizontal or vertical orientation for a given pattern depending on what looks best. The pattern definition itself could also carry a suggestion or default mapping (e.g. a pattern might be marked “radial-friendly”). The system keeps the pattern logic separate from mapping – the same pattern data sent to any device can adapt to that device’s geometry via the selected mapping mode.

Example: Consider a simple chasing light pattern that in 1D moves from LED 0 to LED N in a loop. On a 2D 8x8 panel, if we choose horizontal mapping, the chase will run left-to-right on row 1, then wrap to row 2, etc. If we choose vertical mapping, it will run top-to-bottom on column 1, then next column, and so on. If radial mapping, the chase might start at the center and move outward in all directions symmetrically. All without changing the pattern’s core definition – just the mapping function. This versatility greatly increases the reuse of patterns on different installations.

Real-Time Parameter Control Interface

One of the key features is the ability to tweak patterns live. The pattern language allows **parameters** to be exposed, which the mobile app can adjust in real time. This could be anything from selecting colors to adjusting speeds or toggling certain sub-effects.

Parameter Definitions: In the pattern JSON (or DSL), a `params` section lists all adjustable knobs the pattern provides. Each parameter entry includes: a human-readable name (for the UI), a type, and additional info (range, default, etc.). Supported types include:
- **Boolean toggles** (for on/off features or switching modes).
- **Sliders (numeric)** – typically floats or integers with a min/max range (e.g. speed 0.1–5.0, or quantity of sparkles 0–100).
- **Color pickers** – an RGB or HSV color value. The app might show a color wheel or palette selector. On the device side, this could be represented as a 24-bit RGB value or a palette index.
- **Choice/Enum** – selection among predefined options (e.g. a dropdown to select one of several palettes or waveforms). Represented as an integer index in the device.
- **Gradient/Palette** – possibly a complex type to choose a color palette (if we allow user-defined palettes as parameters). This might be more advanced; simpler is to choose from preset palettes by name or ID.

Runtime Binding: When the app sends a pattern to the ESP32, it also sends the default values for that pattern’s parameters. The interpreter will allocate a small **parameter table** in memory (for example, an array of floats/ints) and store those values. Each cell that depends on a parameter will keep a reference (or index) to the relevant entry in this table. For instance, if a sine wave cell uses the “speed” parameter for its frequency, instead of a fixed number it will store something like `paramIndex = 0` (pointing to params[0] which might be the speed value). During rendering, that cell’s `apply()` function will read the current value from params[0] each time or each frame. This indirection means when the value changes, the effect updates immediately without any reconfiguration needed.

BLE GATT Interface: The BLE protocol will have provisions for the app to **discover parameters** and **send updates**:
- When a pattern is activated, the device can advertise the parameter metadata to the app (or the app already knows from the pattern definition it loaded). The app then populates the control UI accordingly (sliders, toggles, etc.).

- When the user changes a control, the app sends a BLE write or command to update that parameter on the device. For example, a "Set Parameter" command might contain the pattern ID (or an implicit current pattern), the parameter index, and the new value. If using standard BLE, we might have a characteristic per parameter, but since the number of params is dynamic, a simpler approach is a single control characteristic where the app sends a small packet like `[paramIndex, value]`. The device receives it and updates the params table.
- The parameter update takes effect immediately for the next frame. Because the interpreter reads the parameter each use, there's effectively *no latency* beyond BLE transmission. A user sliding a speed control will see the pattern slow down or speed up nearly instantly. If a parameter is very sensitive (e.g. a phase angle), the app might throttle updates or the device might interpolate changes to avoid discontinuities, but typically this isn't an issue (and can be handled in pattern logic if needed).

The parameter interface also allows for external inputs – for example, one could imagine hooking a microphone or sensor to the app and feeding values into parameters (like making a "sound reactive" slider that the app adjusts based on beat detection). The pattern doesn't know the source of the value; it just uses the parameter. This keeps the system flexible and extensible.

Example: Suppose we have a **palette** parameter for a pattern. In the app, the user selects a different color palette. The app sends the new palette ID to the device. The gradient cell in the pattern picks up this new palette ID and uses the corresponding color table for rendering subsequent frames. The lights smoothly transition to the new colors without a restart. Similarly, a **boolean** parameter might turn on/off an overlay: e.g., a "sparkle" toggle that enables a sparkle cell. The sparkle cell's `apply()` could check a boolean and either pass through the input untouched or overlay sparkle highlights. If that boolean is tied to a parameter, toggling it on the app instantly turns the sparkles on or off in the output.

This architecture borrows from systems like Pixelblaze, which expose pattern variables that can be live-tuned. In Pixelblaze's web UI, these correspond to sliders that directly affect the running code. We achieve the same but in a structured way. The **advantage** of our DSL approach is that the app *knows the intended types and ranges* of each parameter (since the pattern definition provides them), so it can present appropriate UI controls. The user doesn't need to know any code – they just see friendly controls to play with the pattern's look. All of this happens without reflashing or restarting the device.

Efficient Pattern Transfer over BLE

Because BLE has limited bandwidth (on the order of a few hundred kilobytes per second at best, often less), pattern definitions must be kept small and transmitted efficiently. Our design ensures patterns are typically only a few KB. Some strategies to optimize transfer and memory:

- **Minimized Format**: The JSON definitions avoid unnecessary bloat. Field names can be abbreviated (e.g., use short keys like `"t":"gradient"` instead of `"type":"gradient"`) or the JSON can be minified (no whitespace). If using a binary representation (like a custom bytecode), the app would translate the pattern into bytes directly. For instance, we might define opcodes for each cell type: e.g., `0x10` = gradient, `0x11` = sineMod, etc., followed by binary data for that cell's parameters. This could compress a pattern significantly. As an example, a gradient cell with two colors might be encoded as: `[0x10, R1,G1,B1, R2,G2,B2]` (7 bytes plus opcode). A text JSON for the same might be 50+ bytes. The trade-off is complexity – implementing a robust binary encoding and decoder on the ESP32. We can start with JSON for clarity and only optimize to binary if needed.

- **Streaming and Acknowledgement**: The BLE protocol should allow sending the pattern in chunks (since BLE characteristic writes are often limited to ~20 bytes without long writes). The device can acknowledge receipt or use a checksum to verify the full pattern. This ensures reliability over BLE, which can occasionally drop packets. Given the small size, this is manageable.
- **On-Device Parsing**: The ESP32 can parse a couple kilobytes of JSON fairly quickly (especially using a small JSON library or a custom parser). However, to minimize device load, the app could do parsing/compiling. For example, the Cordova app (which has more processing power) can convert the pattern JSON into an internal binary structure, then send that to the device. This is what LSSL does: it compiles LED shader code to bytecode in the browser, so the microcontroller doesn't have to [3]. In our case, since patterns are basically data, not full scripts, on-device parsing is not too heavy. But we keep the option open to shift that work to the app if performance dictates.
- **Size Limits**: We will enforce a reasonable maximum pattern size (e.g., 4 KB). This prevents overly complex patterns from using too much BLE time or device memory. Given the modular approach, most patterns remain concise. (Even complex Pixelblaze patterns in code form are often <10 KB [6] as .epe files, and our higher-level format should be even more compact for equivalent functionality since it's not raw code.)
- **Compression**: If needed, the app and device could agree on a simple compression (like gzip or a custom RLE) for pattern payloads. However, with sizes so small, the overhead of compression might not be worth it.

In practice, transferring a 2 KB pattern over BLE should take only tens of milliseconds on a good BLE connection (e.g., 2 KB / 50 KB/s = 0.04 s), although with handshake overhead it might be a couple hundred milliseconds. This is fast enough that the user won't notice a big delay when sending a new pattern. We can further hide transfer latency by pre-loading patterns: for example, if the user selects "download new patterns from server," the app can pre-send them to the device's storage in the background. Then switching patterns on the device is instant.

On-Device Runtime and Virtual Machine

The ESP32 firmware includes a **pattern interpreter** or virtual machine optimized for running these cell-based patterns. Its design priorities are: low memory footprint, speed, and flexibility to support the various cell behaviors. Here's how the runtime is structured:

- **Cell Module Implementations**: For each cell type defined in the language, there is a corresponding C/C++ function (or small set of functions) in firmware that performs that effect. These are highly optimized routines making use of the ESP32's capabilities (e.g., using integer math, approximated math functions, etc., where appropriate). For example, we might have `void applyGradient(const GradientCell &cell, int pixelIndex, Color &ioColor)` which computes the gradient's color at that pixel (overwriting `ioColor`), or `void applySineMod(const SineModCell &cell, int pixelIndex, Color &ioColor, uint32_t t)` that multiplies `ioColor` by a sinusoidal factor. Many cells will utilize common inputs like the pixel's normalized position (which can come from mapping logic) and the global time or frame count for animation. The **timebase** might be a continuously incrementing millisecond counter or a frame counter that the cells use to produce motion (the speed parameters scale how time influences them).

- **Pattern Execution Flow**: When a pattern is loaded or activated, the device **initializes a list of cell objects** in memory. This could be an array or linked list of cell structures, each containing the cell's type and a copy of its parameters (or pointers to parameters table for external params). It also sets up any needed lookup tables or state (e.g., if a cell uses a random generator, it might seed one). At runtime, a main loop (running at the desired frame rate, say 30 or 60 FPS) will iterate over all pixels and apply the cells in sequence, as outlined earlier. This is effectively the **virtual machine** execution: instead of a generic stack machine or register VM (like a CPU emulator), our VM is more static – it's a hardcoded sequence of operations (the cells) that we know in advance. This avoids the overhead of decoding opcodes for every pixel. In essence, we "compile" the pattern to a linked set of C function calls. This design is very efficient on microcontrollers: it leverages compiled code for each operation while still being dynamically configurable. (Pixelblaze takes a similar approach by compiling user code to a bytecode that runs on a custom VM, achieving high frame rates on microcontrollers [7] [2].) Our higher abstraction means we don't expose arbitrary arithmetic to the user, but internally the VM can be thought of as executing a fixed program (list of cell ops) per pixel.
- **Resource Management**: The interpreter uses minimal overhead. Each cell might consume a small struct (a few bytes for type and parameters). The overall pattern perhaps uses a few hundred bytes for all cells and param storage. The VM code itself (the engine that loops and calls cells) is lightweight – essentially a double loop with some pointer chasing. We avoid dynamic memory allocation in the inner loop; any allocation (for cell list, buffers, etc.) is done when a pattern is loaded, not during rendering. The **memory footprint** of the VM core is tiny: an open-source example (LSSL) implemented a similar concept with under 10 KB flash and negligible RAM [4]. Even with many built-in cell types, our code space might be on the order of a few tens of KB, well within ESP32's capacity (which often has >1MB available for app code). Data RAM usage is dominated by the pixel frame buffer (which is `numPixels × 3` bytes for RGB, e.g. 900 bytes for 300 LEDs). Double buffering (if used) would double that, but these numbers are trivial for ESP32's ~520 KB RAM.
- **Optimizations**: We can exploit the ESP32's dual-core nature by dedicating one core to the time-critical task of driving LEDs and computing frames. For example, Core 1 could run the pattern loop at high priority, while Core 0 handles BLE communication and other tasks. This separation ensures that BLE events (or other background tasks) do not stutter the LED output. The LED driving itself (the actual signal output) can be done using the ESP32's RMT peripheral or I²S, allowing DMA to stream data to the LEDs while the CPU prepares the next frame. This is how projects like WLED achieve smooth output. In our system, once a frame's colors are computed in the frame buffer, we trigger the RMT to send it out to the LED strip. While the RMT is sending, we can potentially start computing the next frame if the output is slow (for long strips). If the frame rate is not too high, the CPU might idle or handle BLE commands between frame renders.
- **Frame Timing**: The interpreter can run in a loop with a delay to cap frame rate (to save CPU or to match a desired animation speed). Alternatively, it can be event-driven (compute as fast as possible or whenever a parameter changes significantly). Typically, a stable frame rate (say 50 FPS) is good for visual consistency. The time delta or frame count is passed to cells so that effects move smoothly independent of frame timing.
- **Safety and Stability**: Since patterns are data-defined and run in our controlled VM, we avoid many issues like infinite loops or crashes that arbitrary user code could cause. Each cell's code is known and tested. We can include safeguards (e.g., clamp values, avoid heavy computations if not needed) to ensure no pattern can "brick" the device. In worst case, a pattern could be too complex and slow down the frame rate, but it won't crash – it just yields lower FPS. The user could then choose a simpler pattern. We could also implement a **watchdog** that monitors frame rate and if it drops too

low (say <5 FPS due to an overly heavy pattern with many layers and thousands of LEDs), the device could auto-suspend that pattern or warn the app.

In summary, the on-device runtime functions as a specialized virtual machine for LED graphics. It's optimized to interpret a small set of high-level visual instructions extremely quickly on the ESP32. This approach draws inspiration from Pixelblaze's engine (which compiles expressions to bytecode and runs on a VM) but operates at a higher semantic level. By restricting to known building blocks, we achieve both **performance** and **safety**, as well as simplify the user's job (no coding required to create patterns).

Pattern Storage and Memory Management

ESP32 devices have limited storage, so we need a strategy to manage multiple pattern definitions efficiently. The goal is to allow devices to **cache** a number of patterns locally so they can switch without always requiring a re-transfer from the phone, but also to automatically remove unused patterns to free space.

Storage Medium: We can use the ESP32's flash (external SPI flash) to store patterns. For example, a partition of a few hundred kilobytes could be set aside for pattern storage (in SPIFFS or littlefs, or even as raw binary blobs in NVS). Each pattern, being only a few KB, would occupy a file or record. Storing 100 patterns might use ~200 KB, which is quite reasonable on a 4MB flash chip. The reading of a pattern from flash is fast (a few milliseconds), so loading a pattern on demand is not an issue. We should store the patterns in their **transferred format** (likely JSON or compact binary) for simplicity, and parse into RAM when activating. Alternatively, we could store an already-parsed representation to speed up loading at the cost of a bit more flash usage; however, given the small sizes, parsing is quick enough that it's not necessary to cache the binary representation.

LRU Cache Policy: To ensure we don't run out of storage, the device will implement a **least-recently-used (LRU)** eviction scheme: - We define a maximum number of patterns or total bytes to store (e.g. "cache up to 50 patterns or 200 KB").
- Each time a pattern is received or accessed, mark it with a timestamp or increment a usage counter.
- If adding a new pattern would exceed the cache limit, remove the least recently used pattern(s) until the size is under the limit. We will not remove the *currently active* pattern even if it's LRU – that one must stay. We also might keep a small buffer so we don't thrash (e.g. if near limit, proactively remove a couple of old patterns).
- The metadata (name, ID, last used time) for cached patterns can be stored in NVS or a small index file for quick lookup.

Because patterns can also be shared peer-to-peer and are synced to the server, losing a pattern from a device cache isn't catastrophic – it can be reacquired later if needed. The cache just improves responsiveness (no need to fetch from phone if it's already stored). The mobile app can display which patterns are currently on a device and maybe which are cached vs need downloading.

Memory (RAM) usage: At runtime, only the *active pattern* needs to reside in RAM (aside from a list of cached pattern headers perhaps). So even if many patterns are stored in flash, they don't consume RAM until loaded. When switching patterns, the device will: 1. If the new pattern is in flash (cache), read it into memory; if not, request it from the app via BLE.
2. Parse it and allocate the cell list and param table. Initialize any resources for it.

3. Deallocate the previous pattern's structures (freeing RAM). Possibly keep the previous one if we want instant toggle back (but that's an optimization – simplest is to free it).
4. Begin running the new pattern.

This process might take on the order of 50–100 ms (mostly parsing and allocation), which could produce a slight visible delay when switching patterns. We can mitigate this by doing a double-buffered approach to pattern switch: keep the old pattern running until the new one is fully ready, then swap on the next frame boundary. This way there's no moment of blank output. The switch could even be crossfaded if desired (a nice touch, though not strictly required).

Pattern Size Management: By encouraging patterns to remain small, we avoid clogging memory. If a user tries to create a very large pattern (say by chaining an excessive number of cells), the app can warn or the device can refuse to store it if it breaks the size limit. However, typical use and reasonable safeguards will keep patterns compact.

In terms of **flash wear**, pattern writes are infrequent and small, so even with many updates and sharing, it's not a big concern (ESP32 flash can handle tens of thousands of writes, and patterns would be written occasionally, not constantly).

Pattern Sharing, Evolution, and Syncing

One of the more innovative aspects of the system is treating patterns almost like "living data" that can propagate from device to device, potentially mutate, and eventually sync back to a central repository. This encourages a community-driven ecosystem of patterns. Here's how we facilitate that:

- **Unique Identification**: Every pattern is tagged with a unique ID (and possibly a version or generation number). This ID allows devices and apps to recognize if they already have a given pattern. A convenient approach is to use a hash (SHA-1/MD5) of the pattern content or a GUID assigned by the app when created. A hash doubles as a content identifier to detect modifications (if a pattern is changed, its hash-ID would change, treating it as a new variant).
- **Peer-to-Peer via App**: The mobile app acts as a broker for pattern exchange. If the user has multiple devices connected (or connects to them one after another), the app can sync patterns among them. For example, when Device A is connected, the app can query it for a list of pattern IDs it holds (or at least the currently running pattern's ID plus perhaps a list of cached ones). The app compares that with patterns it knows (in its own library or other connected devices). If it finds an ID that Device A has but Device B doesn't, it can fetch the pattern definition from Device A and send it to Device B. Because the definitions are small, this is fast. The app might even automate this: whenever a new pattern is encountered on any device, it broadcasts it to all devices in the session (assuming user consents). This way, if one device "learns" a new pattern, all the user's devices can learn it.
- **Pattern Evolution**: We can optionally introduce rules for patterns to mutate as they are shared. This could be a fun, experimental feature – for instance, every time a pattern is forwarded via the app, there's a small chance to tweak one of its parameters or swap a cell. Over time, patterns may diverge, creating new variants ("children" of the original). Another way is **user-driven evolution**: the app could allow merging two patterns from the library to form a new one (like combining cells from each, or layering two patterns via an automatically inserted blend cell). The resulting "hybrid" pattern gets a new ID but might record parent IDs in metadata (for lineage tracking). This could be presented as an "evolve" button or a special mode where patterns cycle with slight random changes

(like genetic algorithms breeding new visuals). While not a core requirement for functionality, the architecture is flexible enough to handle this – since a mutated pattern is just another pattern definition. We simply ensure the app treats it as new data and syncs it accordingly.

- **Server Sync**: When the app has internet connectivity, it can upload any new or modified patterns to a central server. The server could be a repository where users share patterns publicly, or it could be the user's private collection. Either way, this cloud backup means no pattern is truly lost even if device caches overflow. For example, if a pattern emerged via device-to-device transfer and the user really likes it, the app will eventually sync it so it can be accessed later or shared with others online. Conversely, the app can download new patterns from the server (say a community library) and then propagate those to the user's devices.

- **Consistency and Conflicts**: With patterns possibly evolving on different "islands" of devices, the server might receive multiple variants of what started as the same pattern. Since each has a unique ID (based on content), they won't overwrite each other inadvertently. The server might maintain metadata like "Pattern X was originally version 1 by user Y, now we have version 1a by user Z". This is more on the software side – our device just sees them as different patterns. The user can manage these via the app (maybe merging or choosing favorites).

- **Security Considerations**: From a security standpoint, pattern sharing is low-risk (it's just LED instructions, not code execution since our VM is constrained to known cells). The worst a malicious pattern could do is maybe flash lights annoyingly or try to overload the LEDs (but we can cap brightness globally to avoid hardware stress). Still, the app could have an option to require user approval before a device receives a pattern from an unknown source. The default might be open to encourage creativity, given the benign nature of patterns.

Use Case Scenario: Imagine Alice designs a new pattern "Sparkle Wave" on her phone (perhaps by adjusting an existing pattern and saving it as a new one). Her phone gives it a new ID and stores it. She applies it to her living room ESP32 device. Later, she visits a friend Bob, and Bob's phone app connects to Alice's device (or Alice's phone connects to Bob's device) – either way, Bob's device doesn't have "Sparkle Wave". The app sees Alice's device has this pattern, so it transfers it to Bob's device through BLE. Now Bob's device lights can also run it. Bob really likes it and tweaks a parameter, saving a variant "Sparkle Wave Fast". That gets a new ID. When Alice reconnects to her device, the app notices Bob's new pattern and syncs it back to Alice's device (and potentially up to the cloud). Over time, many variants of "Sparkle Wave" might exist, but thanks to IDs and syncing, the ecosystem handles them as separate patterns that can all propagate. The server could later show a family tree of how the pattern evolved for curiosity.

From a **system architecture** view, implementing this involves: the device providing a way to list cached pattern IDs (or at least the current pattern ID), the app maintaining a local database of patterns and their IDs (it acts as the knowledge center to decide what to send where), and a simple protocol for the app to request a pattern by ID from a device. We might include in the BLE service a "Pattern Data" characteristic which, when read, returns the currently active pattern's definition. For cached (inactive) patterns, the app might need to explicitly request by ID (which could trigger the device to load it or directly send from flash). Alternatively, for simplicity, the device might only send the active pattern on request; so for sharing, the user would actively play each pattern they want to share so the app can grab it. A more advanced app could iterate through device's patterns if the firmware supports a command to enumerate them. These are implementation details that can be decided based on how transparent we want the process versus user-driven.

Inspirations and Comparisons

It’s worth noting how this design builds on ideas from existing LED control systems while introducing higher abstraction: - **Pixelblaze** (by ElectroMage) demonstrated that an embedded device like ESP8266/ESP32 can compile and run user-created LED patterns in real time using a custom VM, even supporting a web-based live editor [7] [2]. Our system shares the concept of an on-device interpreter for flexibility, but whereas Pixelblaze uses a JavaScript-like scripting approach (the user writes code expressions directly), we move to a **visual/cell-based language**. This raises the abstraction level: users assemble pre-built effects rather than writing math from scratch. This is more user-friendly for non-programmers and also ensures that performance-critical code is pre-optimized in C/C++ for each cell. We also explicitly incorporate multi-device sharing and evolution which are outside Pixelblaze’s scope.
- **WLED** is another popular ESP32 LED controller firmware; it provides a library of predefined effects and an HTTP/JSON API. Users select effects but cannot define new ones on the fly without coding in Arduino. Our architecture can be seen as adding that missing “live programmability” to an ESP32 controller – with the ability to define new patterns on mobile and transmit them without reflashing firmware. We combine the *extensibility* seen in Pixelblaze with the *user-friendliness* (app control, no coding required) seen in WLED.
- **LSSL (LED Strip Shader Language)** by Jeroen Domburg (Spritetm) is an open-source project inspired by Pixelblaze that uses a C-like language compiled to bytecode for LED effects [8] [9]. It offloads compilation to the web browser and runs a tiny VM on the device [3] [4]. This validates our approach of using a bytecode interpreter on ESP32 – it can be made very lightweight and fast. Our difference is LSSL still requires writing shader code; our DSL removes that requirement by providing higher-level primitives.
- **TouchDesigner / node-based visual programming** tools: In the professional lighting and VJ industry, node-based systems allow complex effects by connecting functional blocks. Our pattern language is conceptually similar but tailored for microcontroller limits. We can’t have dozens of heavy nodes due to CPU constraints, but we aim to provide enough primitives to cover rich visual patterns in a constrained environment.

By learning from these systems, we ensure our architecture is both ambitious and realistic. We leverage the ESP32’s strengths (speed, connectivity) and mitigate its constraints (memory, bandwidth) with careful design.

Conclusion

In this report, we outlined a comprehensive architecture for a programmable LED pattern language targeting ESP32 controllers with mobile app control. The design emphasizes modularity (through “cell” building blocks), real-time interactivity, and shareability. Key takeaways include:

- **Compact Pattern DSL**: Patterns are defined in a compact JSON/bytecode format describing a network of visual effect modules. This makes patterns lightweight (a few KB) and easy to transmit over BLE and store on device.
- **Modular Visual Cells**: A library of reusable cells (generators, modifiers, combiners) enables construction of complex animations without writing code. This high-level approach lets users mix and match effects (sine waves, gradients, masks, etc.) much like stacking layers in Photoshop or audio effects in a synthesizer.
- **1D/2D Agnosticism**: Patterns run in an abstract 1D space by default but seamlessly map onto 2D layouts via configurable mapping modes. This ensures one pattern can light up many kinds of LED arrangements appropriately, increasing reuse.

- **Live Parameter Control**: Sliders, toggles, and color pickers tied to pattern parameters allow instant fine-tuning. The BLE-based interface updates the on-device pattern state in real time, so users can interactively adjust the lights and see feedback, enabling a creative "live coding" feel without coding.
- **Efficient ESP32 Runtime**: An optimized interpreter/VM on the ESP32 executes patterns with minimal overhead. By compiling patterns into a sequence of native C function calls for each pixel, we achieve high frame rates even on large LED installations. The runtime's small footprint [4] leaves plenty of headroom on the microcontroller for other tasks (or for more complex patterns).
- **Memory & Caching**: The device caches recently used patterns in flash and evicts old ones via LRU policy to manage limited storage. Only the running pattern uses RAM, ensuring scalability.
- **Pattern Sharing & Evolution**: The mobile app orchestrates pattern distribution between devices. Patterns can spread virally through the user's network of lights, and even evolve into new variants. Eventually, patterns sync to the cloud, creating a living library of light patterns that grows over time.

This architecture offers a powerful platform for creative expression with addressable LEDs. It lowers the barrier to entry (no firmware programming needed to make new patterns) while enabling advanced possibilities (community-driven pattern libraries, generative art evolution, cross-device synchronization). By combining the strengths of prior art (Pixelblaze's real-time engine, WLED's user-friendly control) and adding a new layer of abstraction, the system can cater to both novice tinkerers and experienced light artists. The result is an ecosystem where LED patterns are flexible data – easy to create, share, and transform – bringing dynamic lighting experiences to a broad audience.

With careful implementation and iterative refinement, this LED pattern language and its accompanying system can become a cornerstone for interactive lighting projects, from home décor to large-scale installations, fostering creativity and collaboration in the world of LED art.

Sources:
[2] Pixelblaze's real-time pattern engine runs on a bytecode virtual machine for efficiency, especially on ESP32 (2-2.5× faster than ESP8266).
[1] Pixelblaze uses 16.16 fixed-point math for all numbers to optimize performance on microcontrollers.
[7] Pixelblaze V2 introduced a custom compiler and virtual CPU to achieve higher frame rates and real-time code recompilation on-device.
[4] The open-source LSSL project demonstrates that an LED pattern virtual machine can be implemented in under 10 KB of flash and minimal RAM, showing the feasibility of a lightweight VM on ESP32.

---

[1] [6] Displaying letters - Patterns and Code - ElectroMage Forum
https://forum.electromage.com/t/displaying-letters/932

[2] [5] Don't need the hardware, but would love the software - ElectroMage Forum
https://forum.electromage.com/t/dont-need-the-hardware-but-would-love-the-software/413

[3] [4] [8] [9] GitHub - Spritetm/lssl: Ledstrip Shader Language repo
https://github.com/Spritetm/lssl

[7] Interview on the Embedded.fm podcast - News and Announcements - ElectroMage Forum
https://forum.electromage.com/t/interview-on-the-embedded-fm-podcast/716