Nanolux Developer's Handbook

Version 1.0

5-22-2024

Foreword

This manual is designed to work in tandem with the other methods of documentation. To that end, it attempts to accomplish the goals listed below:

Be Printable

This manual should be under 30 pages, which is printable on consumer printers.

Be Explanatory

This manual should try to explain topics without requiring readers to read excessive amounts of code.

Be Visual

This manual should use images and diagrams to explain topics when text explanations fail.

Be Appealing

This manual should appear presentable and professional.

Clean Code Policy

The original writer of this manual believes two things about clean code:

- Clean code is possible to write.
- Everyone is capable of writing clean code.

Clean code enables developers to write code faster, especially during the typical on boarding process. Clean code is typically modular and easy to configure or modify. Clean code often utilizes elements of object-oriented programming, such as functions and structures.

Questions?

Please direct any questions to Caleb Shilling at the following email address:

cshilling2000@gmail.com

Table of Contents

Terminology Overview

During the 2023-2024 project cycle, new features were implemented that required major changes to both project structure and terminology.

- **Pattern:** An algorithm that controls LED light output for a portion of the LED strip.
- **Strip:** The entire LED strip.
- **Strip Buffer:** Data generated from the previous pattern iteration that will be used to generate the next pattern iteration.
- **Config Data:** Data related to the state of the system.
- **Strip Data:** Configuration data that affects all currently-running patterns. Contains multiple instances of Pattern Data within it.
- **Pattern Data:** Configuration data that affects a single running pattern.
- **Save Slot:** A spot in non volatile storage (NVS) that can store a single strip data instance.

The Nanolux project has many code structures named after the terminology they represent. For example, in the storage.h file, there are 3 structs named "Config_Data," "Strip_Data," and "Pattern_Data."

```
typedef struct{

  uint8_t idx = 0; /// The selected pattern name to run.
  uint8_t brightness = 255; /// The pattern's brightness.
  uint8_t smoothing = 0; /// How smoothed pattern light changes are.
  uint8_t minhue = 0;
  uint8_t maxhue = 255;
  uint8_t config = 0; // diffrent configs
  uint8_t postprocessing_mode = 0; // The current mode for postprocessing

} Pattern_Data;


/// A structure holding strip configuration data.
typedef struct{

  uint8_t alpha = 0; /// How transparent the top pattern is in Z-layering.
  uint8_t noise_thresh = 0; /// The minimum noise floor to consider as audio.
  uint8_t mode = 0; /// The currently-running pattern mode (splitting vs layering).
  uint8_t pattern_count = 1; /// The number of patterns this config has.
  Pattern_Data pattern[PATTERN_LIMIT]; /// Data for all patterns loaded.

} Strip_Data;

/// A structure holding system configuration data.
typedef struct{

  uint8_t length = 60; /// The length of the LED strip.
  uint8_t loop_ms = 40; /// The number of milliseconds one program loop takes.
  uint8_t debug_mode = 0; /// The currently selected debug output mode.
  bool init = true; /// If the loaded config data is valid.
  char pass[16] = ""; // The current device password

} Config_Data;
```

As seen in the above code snippet, Strip_Data contains multiple instances of Pattern_Data. This process is to enable easy LED strip saving and loading, as the only structure that needs to be explicitly transferred to NVS is the running instance of Strip_Data.

Nanolux also uses a "Strip Buffer" structure to store the previous state of the LED strip. This structure holds the buffer used by that pattern along with variables used in calculation, such as the mass and position of a virtual spring. This structure is located in patterns.h.

Each currently running pattern on the strip has an accompanying Strip Buffer object. The relationship between a Strip Buffer object and a Pattern Data object is purely associative: the only piece of information that links them is a common array index.

Nanolux also uses libraries or features of the micro controller that could use additional definition.

- **ESP32:** An Arduino-compatible microcontroller with built-in WiFi and Bluetooth.
- **NVS:** Stands for "non-volatile storage." NVS is persistent memory built into the ESP32 itself that persists through power cycles.
- **FastLED:** An Arduino-compatible library capable of outputting to an LED strip. Contains many useful functions, such as blend().

Data Structure Explanation

Prior to the 2023-2024 capstone cycle, many of the variables and objects defined in the project were defined globally and had only a single instance. To remedy this and to allow expansion of the project's capabilities, many objects were moved into reusable structs. The 5 main structures in the project are:

- Pattern (globals.h)
- Strip_Buffer (patterns.h)
- Strip_Data (storage.h)
- Pattern_Data (storage.h)
- Config_Data (storage.h)

The final 3 structures, defined in storage.h, are utilized for both saving data and loading parameters. This reduces overall complexity as there are no need for additional structures for saving and loading, for example.

The Pattern struct contains the following items:

- index: The index for the given pattern.
- pattern_name: The name of the pattern that is presented to the user on the web app.
- enabled: If the pattern should be presented to the user on the web app.
- pattern_handler: The pointer to the function that generates the pattern.

There is an array of these structs, also in globals.h, that stores every single pattern currently run-able by end users. The description of all these patterns can be seen on the Nanolux wiki. To obtain and run a pattern, use code similar to the following:

```
mainPatterns[manual_pattern_idx].pattern_handler(&histories[0], config.length);
```

The Strip Buffer struct is essentially made up of a variety of objects used by various patterns. Most importantly, it contains an LED buffer. This LED buffer is intended to be used to house the current state of the LED strip. As each running pattern has an associated Strip Buffer object, this means each running pattern has a pattern buffer intended to be used only by it. This logic applies for all other variables associated with a pattern, and also prevents conflicts around multiple patterns utilizing the same variables, such as the position and velocity of a virtual spring.

This is important as some patterns (for example, Hue Trail) utilize the previous state of the LED strip to generate the next state. So, if the current buffer is ever scaled or smoothed, this would mean the next iteration of the pattern would utilize the distorted data instead of the actual previous iteration's LED strip state.

The global Strip Buffer object is defined in main.ino:

```
Strip_Buffer histories[PATTERN_LIMIT];
```

The first of the 3 related storage structures is Strip Data. Strip Data contains visual and audio parameters that apply to the entire strip, including all running patterns. Strip Data also contains a number of instances of Pattern Data.

- **alpha**: How transparent the top layer is when running pattern layering.
- **noise_thresh**: The minimum threshold audio must cross to be considered non-zero during audio processing.
- **mode**: What mode the strip should be running in (strip splitting or pattern layering)
- **pattern_count**: The number of currently-running patterns.
- **pattern**: An array of Pattern_Data structures.

There are 2+ instances of Strip Data across the program, both defined within main.ino. The instance the main loop pulls from is

```
Strip_Data loaded_patterns;
```

There is an additional array of instances defined as:

```
Strip_Data saved_patterns[NUM_SAVES];
```

saved_patterns is the array that is saved to NVS. This process will be described in detail in a later section, but essentially, when the current Strip Data is to be saved to a slot, that particular array position is overwritten with the Strip Data in loaded_patterns. When a save slot is loaded, the inverse happens.

The second storage structure is Pattern Data. Pattern Data contains all data required to run a particular section of the LED strip.

- **idx**: The pattern index that this pattern is currently running
- **brightness**: A scalar applied to the entire pattern to decrease brightness (0-255)
- **smoothing**: How much of the previous LED strip state to include in the final processed output (0-175)

All instances should be contained within loaded or saved patterns.

The final storage structure is Config Data. Config Data stores configuration data such as strip length and target program loop times. It is sometimes referred to as "system settings."

- **length**: The number of LEDs on the current strip.
- **loop_ms**: The number of milliseconds to dedicate to a program loop.
- **debug_mode**: The debug state of the device (serial prints, simulator outputs).
- **init**: Set to 0 if loading from null data. Used to check if non initialized data has been loaded from NVS.

The main Config Data object is defined in main.ino:

```
Config_Data config;
```

Project Structure

Microcontroller

All microcontroller code is located in the repository's "main" folder. The entry point of the program is the file main.ino, which calls all other parts of the microcontroller codebase.

Constants and Globals

There are 2 files dedicated to initializing variables and/or macros: globals.h and nanolux_types.h. Globals contains mostly global variables that aren't utilized in main.ino, in addition to defining the Pattern struct and its accompanying array. nanolux_types contains nearly macros utilized as constant values by the codebase.

Audio Analysis

main.ino calls the audio analysis pipeline, which consists of audio processing functions from the files core_analysis.h and ext_analysis.h. The core_analysis files perform the basic audio analysis that most patterns pull from. The ext_analysis files perform audio analysis processes that are used by only a few patterns, but are more complex.

Utility

The nanolux_util files contain a variety of "helper" functions. The functions in these files are used all throughout the program and deal with hardware and Nanolux-specific custom mathematical operators, among other actions.

Patterns

The files patterns.h and patterns.cpp contain the actual code to run the pattern algorithms themselves. Their function pointers are handled by the Pattern array in globals.h, so this file is rarely directly addressed in the code. These files pull from the palettes.h to get color palettes for a few pattern algorithms.

Storage

The files storage.h and storage.cpp contain the code needed to save and load system setting configurations, patterns, and strip configurations. As stated before, storage.h contains the three data structures used for containing these three data categories.

Networking

Nanolux network communications are managed in the files api.h and webServer.h. The file api.h contains the request handlers for controlling pattern and strip settings, while webServer.h contains the code initializing the web app, bridging to local network connections, starting the built-in router, and manages the API for updating networking-specific device settings. You can find more on the API on its accompanying wiki page.

Web App

All web app code is included in the path WebApp/src. It can roughly be divided into the following segments:

- Encapsulated Components
- Control Components
- Routes
    - Device Settings
    - Wifi Settings
- The API

Encapsulated Components

Encapsulated components are components that

1. Perform a singular function
2. "Encapsulate" functionality of a simpler element (say, a drop down list) so it can be reused without issue

There are far too many to go over here in depth, so here is a list of all encapsulated components:

- Config Drop Down
    - Allows the user to select a configuration for a running pattern (such as volume controlled vs frequency controlled)
- Multi Range Slider
    - Allows the user to specify two values that compose a range
- Numeric Slider

    - Allows the user to enter a value via a slider
- Patterns
    - Presents a drop down list of pattern names supplied elsewhere in the program
- Save Entry
    - Allows the user to save and load a pattern from NVS
- Single Chooser
    - Allows the user to select one or none of of an option list.

Let's use Numeric Slider as an example for the structure of an encapsulated component. It has the following parameters input to the element itself. Here is a snippet of the JSDocs header from the element:

```
/**
 * @brief A UI element that creates a draggable slider and an readout
 * showing the value set by the slider.
 *
 * @param label The label that is displayed alongside the slider.
 * @param min The minimum value selectable by the slider.
 * @param max The maximum value selectable by the slider.
 * @param initial The initial value shown by the slider.
 * @param structure_ref The string reference to store values at.
 * @param update A function to update an external data structure.
 *
 * @returns The UI element itself.
 */
```

The essential values that separate an encapsulated component from a regular component are the structure_ref and update parameters. As seen in a later section, most values are sent to the Audiolux device through data structures, which helps cut down on the number of API calls and makes adding new parameters easier. Effectively, "structure ref" is the element in that structure the component is responsible for modifying. The parameter update contains a function which performs that modification and sets a flag in the parent control component:

```
const valueChanged = async event => {
    current.value = event.target.value;
    update(structure_ref, current.value);
}
```

From the PatternSettings control component:

```
/**
 * @brief Updates a parameter in the pattern data structure with a new value.
 * @param ref The string reference to update in the data structure
 * @param value The new value to update the data structure with
 */
const update = (ref, value) => {
    if(!loading){
        setData((oldData) => {
            let newData = Object.assign({}, oldData);
            newData[ref] = value;
            return newData;
        })
    }
    setUpdated(true);
}
```

Control Components

Control components are responsible for housing a related collection of encapsulated components and coordinating their API connection to the Audiolux device itself. There are currently 3 in the web app:

- Pattern Settings
- Strip Settings
- System Settings

System settings is instantiated separately from the other two, but Strip Settings itself contains an instance of Pattern Settings.

Control components revolve around the concept of a singular data structure. This data structure is updated with new data from the device upon connection, and sends out updates to the device when the user changes a parameter on the web UI. Here is an example of one, taken from Pattern Settings:

```
// Pattern-level data structure
const [data, setData] = useState({
    idx: 0,
    hue_max: 255,
    hue_min: 0,
    brightness: 255,
    smoothing: 0,
```

```
    postprocess: 0,
});
```

As stated earlier, whenever a child encapsulated component causes the data structure to update, the parent control component sets a flag that enables the web app to send an API call to the device. This is controlled by a basic interval that checks for that flag.

Routes / Subpages

The concept of Routes in a web application concept are essentially subpages. There are two routes in the web application at this time: one for controlling device settings and one for controlling WiFi settings.

Device Settings Route

The web page for controlling device settings controls strip, pattern, and system data that does not involve WiFi. It creates the following objects:

- Strip Settings
    - Contains a Pattern Settings object inside of it
- System Settings
- 3 Save Entry objects for the 3 save slots on the device.

WiFi Settings Route

This route needs particular attention for a refactor, and has a lot of repeated code. It is also difficult to understand, so if I were in charge of making changes, I would spend a few hours just unpacking it.

Main Loop Overview

The file main.ino is the entry point for microcontroller code, and as such drives the output LED strip. The file can be roughly split into 5 sections:

1. Initialization
2. Helper Functions
3. Post-Processing
4. Driver Functions
5. Program Loops

This manual section will go over these parts of the main file in order.

Initialization

The first 100 or so lines of the file manage imports, initialize variables, and define critical function headers. These are put into groups of similar variables, and have enough in-line documentation available for an end user. The most critical function here is setup(). This function is immediately ran upon microcontroller boot, and performs the following actions:

- Starts the LED strip.
- Initialize the pin attached to the ESP32's built-in LED.
- Start serial communication.
- Add interrupts for the hardware button.
- Load and verify saves stored in non-volatile storage.

An additional feature of setup() is that it runs the function for starting the web server, which hosts the Nanolux web application. This code is wrapped in an #ifdef macro checking if the macro ENABLE_WEB_SERVER is defined, which allows the web app to be easily disabled.

Helper Functions

The main.ino file has several functions called "helpers" that only main.ino uses. Often, helper functions will either encapsulate code that is used repeatedly throughout the file or encapsulate code that would be clunky to use directly.

calculate_layering()

This function takes two input CRGB array pointers, blurs them together, and places the output into an output CRGB array pointer. Used for both pattern layering and the per-pattern output smoothness parameter.

reverse_buffer()

This function takes in a CRGB array pointer and, for a supplied length, reverses the values contained within. This maps [0] <-> [len-1], This maps [1] <-> [len-2], and so on. This is used inside both postprocessing effects.

unfold_buffer()

The purpose of this function is to take a buffer and a specified length, and "unfold" it to the other side of the buffer. There's some extra math that depends on if the value the length is derived (the actual length of the buffer we care about) is even or odd. In particular, if we performed the even unfolding algorithm in the odd case, there would be an extra pixel at the end of the pattern buffer.

I would highly advise taking a look at this function directly, as its under 10 lines.

print_buffer()

This function simply prints out the values stored within the supplied CRGB buffer to serial. Used for displaying LED output on the simulator.

Post-Processing

The Nanolux codebase currently supports two main methods of postprocessing to the LED strip output. First, the user is able to reverse the LED strip visually. Secondly, the user is able to mirror the pattern around it's center. Both of these postprocessing effects can be applied at the same time as well. These effects are controlled from the function process_pattern().

process_pattern()

The main goal of process_pattern() is to apply both effects without writing excessively long code. The general order of logic in the function can be explained as such:

- If the pattern is to be mirrored, cut the processed length in half.
- If the pattern is to be reversed, reverse the pattern's buffer.
    - This has the effect of "un-reversing" a buffer that is already reversed.
    - This allows the pattern handler function to process the pattern as normal.
- Run the pattern handler.
- If the pattern is to be reversed, reverse the pattern's buffer again.
- If the pattern is to be mirrored, run unfold_buffer() to mirror it.

Driver Functions

The driver functions are low-level functions that perform the logic behind pattern layering or strip splitting, the two main "modes" of the device.

run_strip_splitting()

run_strip_splitting() is the function ran when the Audiolux is in multi-pattern mode. It splits the strip into a number of sections, ranging between 1-4, but the upper limit is configurable through a macro. After calculating the number of pixels to assign per segment, the following loop is ran for each running pattern:

- Run process_pattern() for the pattern currently being addressed.
- Copy the processed segment within the associated Strip Buffer to the first LED strip output buffer.
- Scale the processed segment's brightness within the first LED strip output buffer.
- Blur the smoothed output buffer and the first output buffer together on just the segment allocated to the current pattern.

Once these steps have been ran for each running pattern, the smoothed output buffer is full and ready to be output to the LED strip. Separating the smoothing and brightness adjusting into multiple steps and constraining them to particular segment allows every running pattern to have a different amount of applied smoothing and set brightness.

run_pattern_layering()

Pattern layering takes the first two running patterns and combines them, so that one is running above the other with configurable transparency between the two layers. This currently only works for 2 patterns and any more that are set up do not run.

- Run process_pattern() for both patterns.
- Copy the buffer for each pattern contained in their respective Strip_Buffer object into temporary CRGB arrays.
- Set the brightness of the temporary CRGB array for each pattern.
- Layer the two arrays together.
- Blend the layered CRGB buffer and the smoothed output buffer to smooth the output.

Program Loops

The Nanolux codebase runs most code iteratively: i.e., every 10-200 milliseconds or so, part of the program is run again, allowing each run of the program to build off of the previous run. This allows patterns on the LED strip to change with time, not just input audio. There are two main loops in main.ino: loop() and audio/_analysis(). The latter runs all functions needed to sample audio and pull out features, such as volume and frequency. This loop effectively just runs functions from elsewhere in the program.

loop() is a bit different. It is automatically run repeatedly by the microcontroller (audio_analysis() itself is ran by loop()) and is the "top" of the program. Effectively, this means that all code is ran by and returns to loop, aside from interrupts. Loop is responsible for the following activities:

- Loop timing
    - This ensures all program loops take approximately the same amount of time to complete.
    - At the end of the loop, if not enough time has elapsed, the program will wait until it has.
- Pattern resetting
    - If the web/mobile app has requested a change that requires a reset, all patterns, their strip buffers, and the output buffers will be reset to their initial state.
- Reset button control
    - If the physical button on the Audiolux board is held down for 10 seconds (configurable) the device will reset all saved patterns and configurations.
- Controlling which driver function to run based off user selection
- Displaying to the LED strip

- Outputting to the simulator
- Receiving updates from the web server

The API

The API is covered in a page on the Audiolux wiki, which you can check out here: https://github.com/OPEnSLab-OSU/Nanolux/wiki/Hardware-API

Adding New Patterns

This is a more code-heavy section, so I would recommend brushing up on your C/C++ if you need to.

Adding new patterns to a device running Nanolux is really easy, and doesn't take that much knowledge about programming aside from the basics. This tutorial will navigate the process of adding a pattern that leverages most things you can do with Nanolux.

Stating the Problem

It's often a good idea to precisely state out what you want a pattern to do, in the shortest amount possible. For this demo, we'll be working off of the following statement:

```
Pattern Name: Bar Fill

Brief Description: With either volume or frequency, the lit LEDs will climb with increasing volume or frequency,
and lower if the volume or frequency drops. The hue will be equal to the minimum hue at the lowest pixel and
equal to the maximum hue at the highest one. The brightness and saturation will remain at their maximums.

Configurations: Volume (default), Frequency (1)
```

ESP32 Implementation

There's a couple of rules to follow when actually implementing a pattern:

- Only modify values inside of the strip buffer variable.
- Don't hold up the program (no calls to delay()).
- Only access positions inside the LED buffer that are within the length you are allowed to modify (the function parameter "len").

Let's start with defining the basic structure of our pattern (in patterns.cpp). Don't forget to create the header function in patterns.h!

```
#define VOLUME 0
#define FREQUENCY 1
/// @brief Displays a pattern that occupies "lower" pixels at lower values,
/// and "higher" pixels at higher values.
/// @param buf Pointer to the Strip_Buffer structure, holds LED buffer and history variables.
/// @param len The length of LEDs to process
/// @param params Pointer to Pattern_Data structure containing configuration options.
void bar_fill(Strip_Buffer * buf, int len, Pattern_Data* params){

  switch(params->config) {

    case VOLUME: default: {


      break;
    }
    case FREQUENCY: {


      break;
    }
  }
}
```

This is a good "clean" initial structure that doesn't go overboard with extra code. Cases are clearly labeled using macros, and the Doxygen header is present describing the pattern. You can check out the final pattern in patterns.cpp if you wish to see the final version.

Next, we need to add the pattern to the global patterns array. This is located in globals.h,

```
Pattern mainPatterns[]{
    { 0, "None", true, blank},
    { 1, "Pixel Frequency", true, pix_freq},
    { 2, "Confetti", true, confetti},
    { 3, "Hue Trail", true, hue_trail},
    { 4, "Saturated", true, saturated},
    { 5, "Groovy", true, groovy},
    { 6, "Talking", true, talking},
    { 7, "Glitch", true, glitch},
    { 8, "Bands", true, bands},
    { 9, "Equalizer", true, eq},
    { 10, "Tug of War", true, tug_of_war},
```

```
    { 11, "Rain Drop", true, random_raindrop},
    { 12, "Fire 2012", true, Fire2012},
};
int NUM_PATTERNS = 13;
```

And we need to make the following change:

```
Pattern mainPatterns[]{
    { 0, "None", true, blank},
    ...
    ...
    { 12, "Fire 2012", true, Fire2012},
    { 13, "Bar Fill", true, bar_fill},
};
int NUM_PATTERNS = 14;
```

The ESP32 code is ready to run the pattern, but the web app needs a minor change as well.

Web App Implementation

The change is pretty simple: we need to add the pattern configurations (volume, frequency) into the web app. Navigate to the file at the relative path WebApp/src/components/config_drop_down/index.js and find the following object:

```
const configs = [
    ["None"],
    ...
    ...
    ["Default"]
]
```

You'll need to add the new pattern's configurations at the end of this list, which is the list of configurations for all patterns.

```
const configs = [
    ["None"],
    ...
    ...
    ["Default"],
    ["Volume", "Frequency"]
]
```

If you followed this guide step-by-step, your pattern should now be selectable in the web app!

Building the Nanolux Developer's Handbook

The file manual_builder.py is a neat script used to build the Nanolux Developer's Handbook from the markdown files present in the markdowns folder.

Dependencies

Use the following commands to install all dependencies:

pip install markdown

pip install pdfkit

pip install pypdf

You'll also need the file wkhtmltoppdf.exe from python-pdfkit. Get it from the 7z Archive Windows download at this link: https://wkhtmltopdf.org/downloads.html

Add this file to this folder. It is already in the .gitignore so it isn't committed to the repository.

This utility is only validated to run on Windows as of 5/22/2024.

Adding New Sections

To add a new section to the manual, move the markdown file to the markdowns folder. Then, add the file name (omit the .md extension) to the "input_filenames" list in manual_builder.py:

```
input_filenames = [
    'title_0',
    'foreward_1',
    'toc_2',
    'terminology_3',
    'structs_4',
    'project_struct_5',
    'main_loop_6',
    'add_patterns_7',
    'building_manual_8',
    'changelog_9'
]
```

If inserting a section requires renaming others, change the numbering in both the python file and on all markdown files. Make sure to update the front cover (release date, version) and the changelog section.

Changelog

v1.0

Released 5/22/2024

- Added the section "Adding New Patterns"
- Added the section "Building the Nanolux Developer's Handbook"
- Added the section "Changelog"
- Added the manual_builder.py script for building the manual and the accompanying readme file