Nanolux Developer's Handbook

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.
- **History:** 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;
  uint8_t brightness = 255;
  uint8_t smoothing = 0;

} Pattern_Data;

typedef struct{

  uint8_t alpha = 0;
  uint8_t noise_thresh = 0;
  uint8_t mode = 0;
  uint8_t pattern_count = 1;
  Pattern_Data pattern[NUM_PATTERN_TYPES];

} Strip_Data;

typedef struct{

  uint8_t length = 60;
  uint8_t loop_ms = 40;
  uint8_t debug_mode = 0;
  bool init = true;

} 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 "Pattern History" 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 Pattern History object. The relationship between a Pattern History 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)
- Pattern_History (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 Pattern History 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 Pattern History 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 history, 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 Pattern History object is defined in main.ino:

```
Pattern_History histories[NUM_SUBPATTERNS];
```

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.

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

TBD

Main Microcontroller Project Loop

The main.ino file is the entry point for the Nanolux microcontroller code. As such, it has many jobs, such as managing program loop length times, running the LED strip, processing patterns, and running the audio analysis pipeline. Many of these jobs are outsourced to other files, but have their main call in main.ino. This file also initializes all global variables that have their first use in main.ino.

setup():

The setup function is ran once upon device power up. It has a few roles:

- Initializing pin output for the built in LED and external LED strip
- Beginning Serial output
- Loading and verifying saves from the NVS
- Starting device reset timer functionality

run_strip_splitting():

The strip splitting function is the backbone of the Nanolux main loop. It contains the ability to both run a singular pattern and split the strip into multiple separate patterns.

The function first calculates the number of pixels per running pattern. Then, for each running pattern: