Debugging Board

Lil gameboy doodad

Overview

The debugging board is a **dedicated auxiliary system** whose only job is to make the rest of the robot less painful to work with.

> It is not part of the rover’s core functionality.

Purpose

At a high level, the debugging board serves roles:

Visibility

Provide real-time insight into system state:

- logs
- status indicators
- network activity
- subsystem health

Instead of digging through serial output on multiple MCUs or adding temporary debug code everywhere, this board aggregates and presents useful information.

Control / Interaction

> Todo :D

Isolation of debugging concerns

> Todo :D

Physical components

The exact hardware may evolve, but the debugging board generally consists of:

Ethernet interface

- Connects to the system network (switch / internal bus)
- Receives and sends packets (including protobuf-based messages)
- Acts as a bridge between the debugging interface and the rest of the robot

Display

- Shows system state, logs, or selected information
- It is a **ILI9341 SPI Display**

Used for quick, local feedback without needing a laptop.

Input interface (buttons / panel)

- Physical buttons or switches
- Used to:
  - trigger actions
  - navigate menus
  - send commands

Display - ILI9341 Hardware Configuratoin

The debugging board incorporates a graphical display based on the ILI9341 controller. This display serves as the primary local interface for presenting system state, diagnostics, and user feedback.

The ILI9341 is a widely used TFT LCD controller that integrates display driving logic, internal GRAM (Graphics RAM), and a command-based interface over serial or parallel buses. In this system, it is used in **SPI mode**, which aligns with the board's pin constraints and simplifies integration with the MCU.

Functional Role in the System

Within the debugging board, the display is responsible for:

- Rendering system status (connectivity, subsystem health, etc.)
- Displaying structured debugging information
- Providing immediate visual feedback to user input (button interactions)
- Supporting simple UI constructs (menus, indicators, overlays)

> The display is not intended for high-throughput graphics or complex rendering. Its role is **informational and interactive**, not graphical-intensive.

Features of the ILI9341

The ILI9341 controller provides a set of features well suited for embedded applications.

Resolution and Color Depth

- Resolution: **240 × 320 pixels**
- Color depth: **16-bit RGB (RGB565)**

This provides sufficient resolution for:

- text rendering
- simple UI layouts

- basic graphical elements (icons, shapes)

Internal GRAM (Frame Buffer)

> The controller includes internal **Graphics RAM (GRAM)**, which stores pixel data.

- The MCU does **not** need to maintain a full framebuffer
- Pixel data is written directly to the display over SPI
- The display retains the image until overwritten

This significantly reduces RAM requirements on the MCU, which is critical in embedded systems.

Command-Based Interface

The display is controlled through a command/data protocol:

- Commands configure behavior (e.g., orientation, pixel format)
- Data writes update pixel values in GRAM

Typical operations include:

- setting an address window
- writing pixel data
- issuing initialization sequences

Display Orientation and Addressing

The controller supports:

- configurable screen rotation (portrait / landscape)
- programmable address windows

This allows:

- flexible UI layout
- efficient partial updates (writing only specific regions)

Hardware Reset and Initialization

The display requires:

- a hardware reset sequence
- a series of configuration commands during initialization

These typically configure:

- power control
- gamma curves
- pixel format
- memory access control

> ILI9341 is a relatively complex and if you want to do anything with the internal library of it you need more than what can be written here. **Read the** official documentation

MCU Configuration

The SPI peripheral must be configured with:

- Mode: **Full-Duplex Master**
- Data size: **8-bit**
- First bit: **MSB-first**
- Clock polarity: **Low**
- Clock phase: **1st edge**
- NSS: **Software**
- Baud rate prescaler: selected based on display stability

These settings must match the display’s timing requirements.

> For the baud rate, you want it to be as high as possible without it being unstable. For debugging and testing, it's good practice to lower it first, get it working there (as it is a lot more stable) and then increase it again.

Display - ILI9341 Library

Purpose

The `ili9341` library provides the low-level and mid-level drawing interface for the ILI9341-based display used on the debugging board.

Its role is to hide the raw command sequence and SPI transaction details of the display controller behind a set of functions for:

- initialization
- display configuration
- pixel and region drawing
- primitive graphics
- text rendering
- monochrome bitmap rendering
- rounded rectangle rendering

> In other words, this library is the software layer that turns the display from a peripheral into a usable rendering surface.

Scope of the Library

This library sits close to the hardware.

It is responsible for:

- driving the ILI9341 controller over SPI
- controlling the display GPIO lines (`CS`, `DC`, `RST`)
- issuing the controller initialization sequence
- writing pixel data to the display GRAM
- exposing simple drawing primitives for higher-level UI code

It is **not** responsible for:

- application UI logic
- layout management
- widget systems
- maintaining a full framebuffer

- asynchronous rendering scheduling

> This is a direct-draw display driver and utility library, not a graphics framework.

High-Level Design

The library is structured around four layers of functionality.

Transport layer

These functions send commands and bytes over SPI:

- `ILI9341_SPI_Send()`
- `ILI9341_Write_Command()`
- `ILI9341_Write_Data()`

Display control layer

These functions manage display state and configuration:

- `ILI9341_Reset()`
- `ILI9341_Set_Address()`
- `ILI9341_Set_Rotation()`
- `ILI9341_Enable()`
- `ILI9341_Init()`

Primitive drawing layer

These functions draw directly to the screen:

- single colours
- pixels
- colour bursts
- lines
- rectangles
- bitmaps
- colour arrays

Utility rendering layer

These functions build on the primitives to provide:

- text rendering
- rounded-corner rendering
- custom monochrome corner bitmap generation

This layered structure is important. Most higher-level code should use the drawing primitives and utility functions, not manually emit ILI9341 commands unless there is a very specific reason.

Hardware Interface Definitions

The header defines the display connection through compile-time macros.

SPI instance

```
#define HSPI_INSTANCE &hspi1
```

This selects the SPI peripheral used to communicate with the display.

GPIO control lines

```
#define LCD_CS_PORT TFT_CS_GPIO_Port
#define LCD_CS_PIN TFT_CS_Pin

#define LCD_DC_PORT TFT_DC_GPIO_Port
#define LCD_DC_PIN TFT_DC_Pin

#define LCD_RST_PORT TFT_RESET_GPIO_Port
#define LCD_RST_PIN TFT_RESET_Pin
```

These define:

- chip select
- data/command selection
- hardware reset

The library assumes these symbols are provided by the board support layer.

Screen dimensions

```
#define ILI9341_SCREEN_HEIGHT 240
#define ILI9341_SCREEN_WIDTH 320
```

These define the nominal physical display dimensions..

Burst limit

```
#define BURST_MAX_SIZE 500
```

This controls the maximum temporary buffer size used during burst-style SPI transfers.

It affects:

- solid colour fills
- colour array streaming
- bitmap rendering

This is a performance and stack/RAM tradeoff parameter.

Color Definitions

The header provides a set of named RGB565 color constants, for example:

- `BLACK`
- `WHITE`
- `RED`
- `GREEN`
- `BLUE`
- `YELLOW`
- `CYAN`
- `MAGENTA`

These are convenience values for application code and drawing functions.

All colors are represented in **16-bit RGB565 format**, which matches the configured pixel format of the display controller.

Initialization Sequence

`ILI9341_Init()`

```
void ILI9341_Init(void);
```

This is the main initialization routine.

What it does

It performs:

1. display enable
2. SPI init hook
3. hardware reset
4. software reset
5. a full controller configuration sequence
6. exit from sleep mode
7. display on
8. initial screen rotation selection

Initialization sequence contents

The function writes a fixed command sequence configuring:

- power control
- driver timing
- pump ratio
- VCOM control
- memory access control
- pixel format
- frame rate
- gamma correction
- sleep exit
- display enable

This is the board’s current known-good configuration for the display.

Why this matters

This sequence is not arbitrary boilerplate. It defines the electrical and visual behavior of the panel.

If it is modified, the maintainer must understand whether the change is:

- controller-required
- panel-specific
- timing-related
- cosmetic
- or cargo-culted from another project

Basic Drawing Primitives

`ILI9341_Draw_Colour()`

```
void ILI9341_Draw_Colour(uint16_t Colour);
```

Writes one pixel’s worth of RGB565 data to the display.

This function assumes the correct address window is already set.

It is mainly an internal low-level helper.

`ILI9341_Draw_Colour_Burst()`

```
void ILI9341_Draw_Colour_Burst(uint16_t Colour, uint32_t Size);
```

Draws a repeated color value over a number of pixels.

Use case

Efficiently fill:

- large solid regions
- lines
- screen clears

How it works

It creates a temporary burst buffer containing repeated color bytes and transmits it in chunks.

> This is much more efficient than sending each pixel individually.

Importance

This function is central to the performance of:

- full screen fills
- rectangle fills
- line drawing

`ILI9341_Draw_Colour_Array()`

```
void ILI9341_Draw_Colour_Array(const uint16_t *Colour, uint32_t PixelCount);
```

Draws an array of RGB565 pixel values.

Use case

Use this when the caller already has pixel data prepared, for example:

- image rendering
- precomputed graphics
- generated color buffers

Important implementation detail

> The function converts each `uint16_t` color into big-endian byte order before sending.

This is correct for SPI transmission to the display controller.

`ILI9341_Draw_Pixel()`

```
void ILI9341_Draw_Pixel(uint16_t X, uint16_t Y, uint16_t Colour);
```

Draws one pixel at a specific coordinate.

Behavior

It:

- bounds checks the coordinate
- manually sets X address
- manually sets Y address
- issues memory write
- writes one pixel color

Performance note

> This is a **very slow** operation compared to region-based drawing because it reissues addressing commands for every pixel.

It is suitable for:

- sparse pixel updates
- debugging
- very small shapes

It is not suitable for rendering larger regions.

`ILI9341_Fill_Screen()`

```
void ILI9341_Fill_Screen(uint16_t Colour);
```

Fills the whole display with one color.

Behavior

It sets the address window to the whole screen and then sends a repeated-color burst.

Text Rendering

`ILI9341_WriteString()`

```
void ILI9341_WriteString(uint16_t x, uint16_t y, const char *str,
                         ILI9341_FontDef font, uint16_t color,
                         uint16_t bgcolor);
```

Renders a null-terminated string using the specified font and foreground/background colors.

Behavior

- iterates through each character
- wraps to the next line if the current X position exceeds screen width
- stops if the next line would exceed screen height

Internal helper

This uses the internal function:

```
static void ILI9341_WriteChar(...)
```

which renders one character pixel-by-pixel using the font bitmap.

Bitmap Rendering

`ILI9341_Draw_Bitmap()`

```
void ILI9341_Draw_Bitmap(uint16_t x, uint16_t y,
                         uint16_t w, uint16_t h,
                         const uint8_t *bitmap,
                         uint16_t Color, uint16_t BgColor);
```

Draws a **1-bit-per-pixel bitmap** into a rectangular region.

Expected bitmap format

The input bitmap is interpreted as packed monochrome data:

- 1 bit per pixel
- row-major
- MSB-first within each byte

Rendering behavior

For each bit:

- set bit -> draw `Color`
- clear bit -> draw `BgColor`

Use case

This is useful for:

- icons
- glyph-like shapes
- masks
- rounded corner patterns

> It is not for full-color image rendering.

R³: Rounded Rectangle Rendering

The library includes support for rounded rectangle outlines using generated monochrome corner bitmaps.

>

> This is more advanced than the rest of the primitive API and deserves separate explanation.

Why?

> The reasons why the R³ system is highly important - if not necessary - are plenty and extensive. That's why I compiled a pastebin that includes all reasons. Feel free to read it even though I believe it is pretty self explanatory

Concept

A rounded rectangle is rendered by:

1. generating a 1bpp bitmap for one rounded corner
2. rotating that bitmap to obtain all four corners
3. drawing the four corner bitmaps
4. drawing straight rectangle segments between them

This is a practical method for an SPI-driven display because it avoids expensive per-pixel circle calculations at draw time for every corner.

Internal helpers

The implementation includes internal static helpers:

- `ILI9341_Get_Rounded_Corner_Bitmap()`
- `bitmap_rotate_90_cw_1bpp()`
- `ILI9341_Build_All_Rounded_Corners()`
- `ILI9341_Draw_Rectangle_Custom_Corner()`

These are not part of the public API, but they are important for maintainers to understand.

`ILI9341_Draw_Rectangle_Rounded_Corner()`

```
result_t ILI9341_Draw_Rectangle_Rounded_Corner(
    uint16_t X, uint16_t Y, uint16_t Width, uint16_t Height,
    uint8_t thickness, uint8_t radius,
    uint8_t *corner_buffer, size_t corner_buffer_size,
    uint16_t Colour, uint16_t Bg_Colour);
```

This is the main public rounded rectangle API currently implemented with explicit caller-provided corner buffer storage.

Why caller-provided memory is used

The function requires the caller to provide a temporary buffer for the generated corner bitmaps.

This avoids hidden dynamic allocation and gives the caller control over memory use.

Buffer sizing

> The function expects enough memory for **four** 1bpp bitmaps, one for each corner.

It computes the required size as:

```
4 * (((radius + 7) >> 3) * radius)
```

in bytes.

Return values

- `RESULT_OK` on success
- `RESULT_ERR_NO_MEM` if the provided buffer is too small
- `RESULT_ERR_INVALID_ARG` for invalid parameters via internal helpers

Use case

This function is appropriate when the UI wants rounded bordered rectangles without a full framebuffer.

Menu Driver - Overview

Purpose

> The menu driver is a **page-based UI framework** for an embedded display (ILI9341).

It defines:

- how UI is structured into pages
- how state is stored per page
- how navigation works
- how rendering is organized

Architecture Position

```
[ Application Logic ]
↓[ Menu Driver ]
↓[ ILI9341 Driver ]
↓[ SPI / Hardware ]
```

It **does not**:

- own the main loop
- schedule tasks
- interpret input fully

It **does**:

- define UI structure
- manage page lifecycle
- coordinate rendering

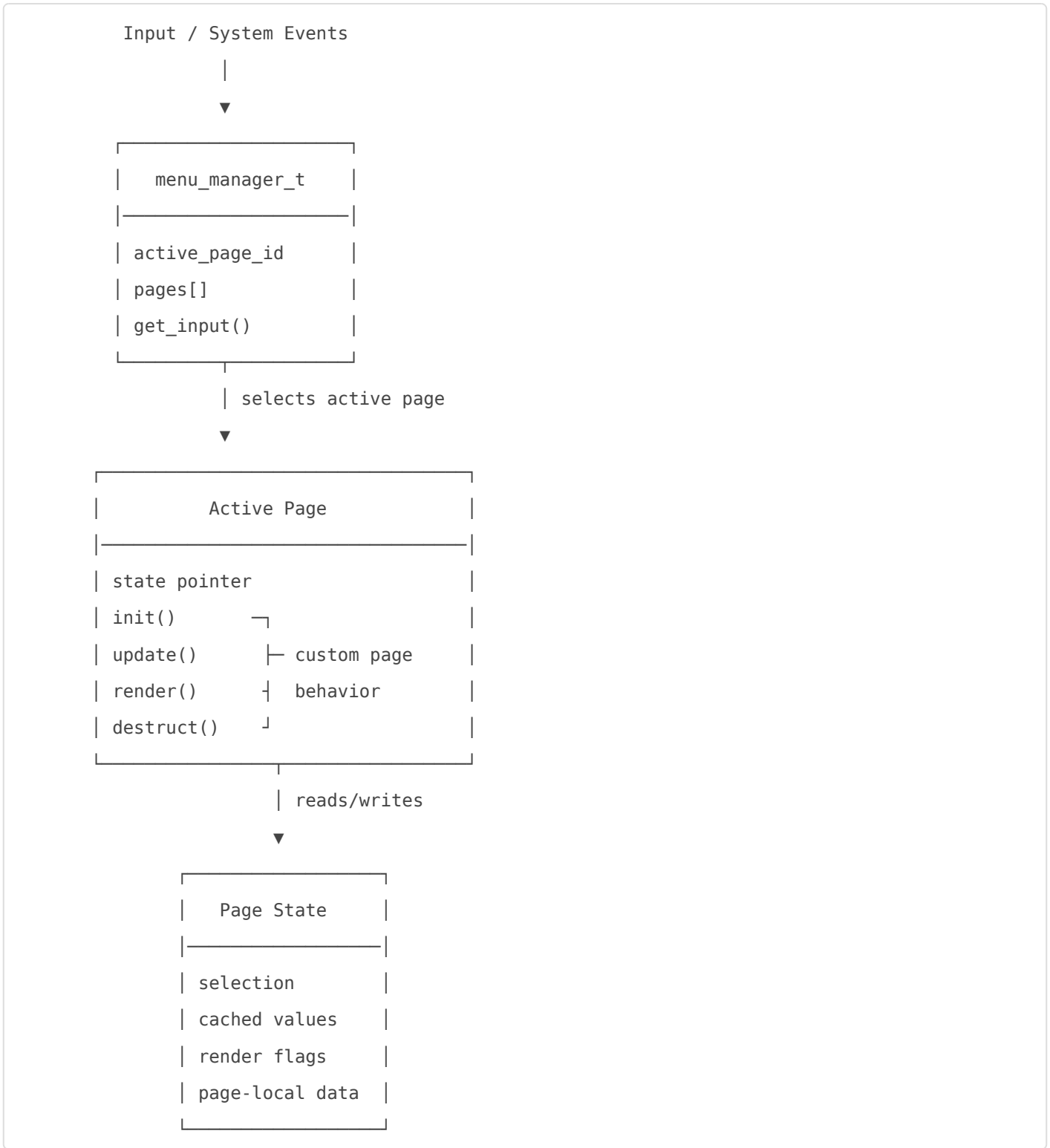
1.3 Design Model

Everything revolves around:

> “A UI is a collection of pages with lifecycle and state.”

Each page has:

- state
- init/update/render/destruct
- parent relationship



Menu Driver - Configuration Layer

Visual Configuration

```
#define MENU_DRIVER_BACKGROUND_COLOR 0x0000
#define MENU_DRIVER_FOREGROUND_COLOR 0xFFFF
```

Black background, white foreground.

Layout Constraints

```
#define MENU_SIDEBAR_WIDTH 38
```

Sidebar (ribbon) width.

Capacity Limits

List Pages

```
#define MAX_LIST_ENTRIES 10
#define MAX_LIST_TITLE_LEN 24
```

Overview Pages

```
#define MENU_OVERVIEW_MAX_ENTRIES 10
#define MENU_OVERVIEW_MAX_ENTRY_TITLE_LEN 12
```

Global

```
#define MAX_PAGE_NAME_LEN 20
```

These define:

- memory footprint
- UI density
- rendering assumptions

Menu Driver - Core Data Structures

Page State Types

A *page state* is:

> The **persistent data container** that represents everything a UI page needs to function between frames.

Not just data. It’s:

- memory of what the user did
- memory of what was rendered
- memory of external data (diagnostics, etc.)

List Page State

```
typedef struct {
  uint8_t num_entries;
  uint8_t selected_index;
  uint8_t entry_ids[MAX_LIST_ENTRIES];
  const uint8_t (*entry_icons)[MENU_DRIVER_ICON_BYTE_SIZE];
  bool first_render;
} page_list_state;
```

Responsibilities:

- track selection
- map entries → page IDs
- hold icons
- manage first render optimization

Overview Page State

> Todo :D

State Union

```
typedef union {
  page_list_state list;
  page_overview_state overview;
} menu_page_state;
```

Page Type

```
typedef enum {
  MENU_PAGE_TYPE_LIST,
  MENU_PAGE_TYPE_OVERVIEW,
} menu_page_type_t;
```

Used to interpret the union correctly.

Page Object

```
typedef struct {
  menu_page_state *state;
  menu_page_type_t type;
  unsigned char id;
  unsigned char parent_id;
  bool needs_render;

  char name[MAX_PAGE_NAME_LEN];

  void (*init)(menu_page_state *);
  void (*update)(menu_manager_t *);
  void (*render)(menu_manager_t *);
  void (*destruct)(menu_page_state *);
} menu_page_t;
```

> This is the **core abstraction**.

Definition and Role

A page object represents one logical screen within the menu system. It encapsulates:

- the data required to represent the page (via its state)
- the functions required to manage its lifecycle
- metadata used for navigation and identification

This abstraction allows the menu system to treat all pages uniformly, regardless of their internal implementation or purpose.

Important Fields

Render Control

```
bool needs_render;
```

This flag indicates whether the page requires re-rendering.

It allows the system to avoid unnecessary redraw operations, which is critical in environments where display updates are expensive.

The responsibility for managing this flag lies with the **page implementation**.

Lifecycle Function Pointers

Each page defines its own behavior through four function pointers:

Initialization

```
void (*init)(menu_page_state *state);
```

Responsible for preparing the page state when the page becomes active.

Typical responsibilities include:

- resetting selection indices
- initializing flags
- preparing any required data structures

Update

```
void (*update)(struct menu_manager_t *manager);
```

Handles input processing and state updates.

This function is expected to:

- read input through the manager
- modify internal state accordingly
- trigger page transitions if necessary

Render

```
void (*render)(struct menu_manager_t *manager);
```

Responsible for drawing the page to the display.

This function should:

- read from the page state
- issue drawing commands via the display driver
- respect the `needs_render` flag when applicable

Destruction

```
void (*destruct)(menu_page_state *state);
```

Handles cleanup when the page is no longer active.

In embedded systems, this typically involves:

- resetting state fields
- releasing logical ownership of resources

Dynamic memory cleanup is generally not required unless explicitly used.

Menu Manager

```
typedef struct {
  unsigned char active_page_id;
  const menu_page_t *pages;
  menu_input (*get_input)(void);
} menu_manager_t;
```

Responsibilities:

- track active page

- provide input access
- hold page table

Does NOT:

- validate anything
- own memory
- manage concurrency

Menu Driver - Overview Page

Introduction

The **List Page** is a navigation-oriented page type within the menu driver. It provides a structured interface for selecting between multiple entries, typically representing:

- subpages
- actions
- system modules

It is the primary mechanism for **user-driven navigation** within the menu system.

Purpose

The list page exists to answer:

> *“Where do you want to go next?”*

It is not responsible for displaying system state in detail. Instead, it:

- presents a bounded set of selectable entries
- tracks the current selection
- provides visual feedback for navigation
- enables transitions to other pages

In practice, it functions as the **entry point and routing layer** of the UI.

Architectural Role

The list page sits at the intersection of:

- **input handling** (user navigation)
- **menu structure** (page hierarchy)
- **visual rendering** (icons and labels)

```
[ Input ] → [ List Page ] → [ Page Transition ]
```

It does not consume system data (like overview pages), but rather controls **flow through the interface**.

Data Model

The list page is backed by the following state structure:

```
typedef struct {
  uint8_t num_entries;
  uint8_t selected_index;
  uint8_t entry_ids[MAX_LIST_ENTRIES];
  const uint8_t (*entry_icons)[MENU_DRIVER_ICON_BYTE_SIZE];
  bool first_render;
} page_list_state;
```

Entry Management

`num_entries`

Defines how many entries are currently active.

This value must not exceed `MAX_LIST_ENTRIES`.

`entry_ids`

Maps each visible entry to a logical identifier.

These IDs are typically used to:

- determine which page to switch to
- associate actions with selections

`entry_icons`

Pointer to icon data associated with each entry.

- Icons are rendered alongside entries
- Each icon is a fixed-size bitmap
- Icons are stored in flash as static data

Selection State

`selected_index`

Indicates which entry is currently selected.

This is the central piece of state for navigation.

All rendering and transitions depend on this value.

Render Control

`first_render`

Indicates whether the page is being rendered for the first time.

Used to:

- trigger full initial draw
- avoid redundant rendering of static UI elements

Rendering Model

The list page uses a **focused rendering strategy**, rather than displaying all entries simultaneously.

Visible Entries

Only three entries are rendered at any time:

- previous entry
- current (selected) entry
- next entry

This creates a scrolling effect without requiring full list rendering.

Rendering Optimization

The only expensive draw of the list page is the initial one which draws the selection border, all initial entries (both icons and names).

After that the only thing that gets redrawn are the icons and the texts. There is also heavier optimization done for minimal font redrawing by keeping track of previously rendered text widths.

> This is critical for SPI-driven displays, where bandwidth is limited.

Interaction Model

The list page assumes an abstract input interface:

```
menu_input (*get_input)(void);
```

The page does not interpret physical inputs directly. Instead, it operates on abstract input values, allowing it to remain independent of hardware specifics.

Expected interactions include:

- move selection up
- move selection down
- confirm selection

Relationship to Menu System

The list page enables hierarchical navigation through:

- `entry_ids` → target page identifiers
- `parent_id` (in `menu_page_t`) → upward navigation

> This allows the menu system to behave as a **tree of pages**, rather than a flat structure.

Performance Considerations

The list page is designed for constrained environments:

- partial rendering minimizes SPI usage
- static memory avoids allocation overhead
- limited visible entries reduce draw complexity