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.

Revision #4
Created 2026-04-15 11:17:17 UTC by Nikolaos Diamantopoulos
Updated 2026-04-16 08:27:10 UTC by Nikolaos Diamantopoulos