Structuring an Arduino Display Driver (WS0010)

Creating a "driver" for an embedded system like an Arduino, especially for a display, typically means developing an Arduino Library. This standardized structure makes your code reusable, easy to install, and understandable by others.
You're already off to a great start with your WS0010.h and WS0010.cpp files! The next steps involve organizing them within a specific directory structure and adding a few crucial metadata files.
Here's a breakdown of the standard layout and what to include:

1. The Core Library Directory Structure

An Arduino library follows a specific folder hierarchy:

```
YourLibraryName/
├── src/
│   ├── YourLibraryName.h
│   └── YourLibraryName.cpp
├── examples/
│   └── BasicDisplay/
│       └── BasicDisplay.ino
├── keywords.txt
├── library.properties
└── README.md (Optional, but highly recommended)
```

Key Points:

- **YourLibraryName/**: This is the main directory for your library. The name should be descriptive and generally match the main class name inside. For your case, it could be WS0010_OLED or WS0010_Display.
- **src/**: This folder contains all your actual source code (.h and .cpp files).
- **examples/**: This folder holds subfolders, each containing a runnable Arduino sketch (.ino file) demonstrating how to use your library.
- **keywords.txt**: Used by the Arduino IDE to highlight specific keywords from your library.
- **library.properties**: A mandatory file containing metadata about your library.
- **README.md**: A Markdown file providing general information about your library, installation instructions, usage, and examples.

2. Header File (WS0010.h - now potentially YourLibraryName.h)

This file defines the public interface of your driver. It tells users what functions and variables

are available to them.

```
#ifndef WS0010_DISPLAY_H
#define WS0010_DISPLAY_H

#include <Arduino.h> // Essential for Arduino specific types like 'byte', 'pinMode', etc.

// Define constants for display dimensions or other configurations
#define WS0010_WIDTH  128
#define WS0010_HEIGHT 64

// If your display uses specific pins, you might define them here
// Or, allow them to be passed in the constructor.
// #define DATA_PIN_D0 2
// ...

// Enum for display commands (example)
enum WS0010_COMMANDS {
  DISPLAY_ON_CMD = 0xAF,
  SET_PAGE_ADDRESS = 0xB0,
  // ... more commands
};

class WS0010_Display {
public:
  // Constructor: Initialize the display object.
  // It should take parameters for the pins connected to the display.
  // Example for an 8-bit parallel interface:
  WS0010_Display(int d0_pin, int d1_pin, int d2_pin, int d3_pin,
                 int d4_pin, int d5_pin, int d6_pin, int d7_pin,
                 int rs_pin, int rw_pin, int en_pin, int cs1_pin, int cs2_pin);

  // Initialization method: Sets up display hardware.
  // This is typically called in setup()
  void begin();

  // High-level drawing functions (Graphics Mode)
  void clearDisplay();
  void setPixel(int x, int y, bool color = true); // true for ON, false for OFF
  void drawLine(int x0, int y0, int x1, int y1, bool color = true);
  void drawRectangle(int x, int y, int w, int h, bool color = true, bool filled = false);
  void drawCircle(int x0, int y0, int r, bool color = true, bool filled = false);
  void drawBitmap(int x, int y, const unsigned char *bitmap, int w, int h, bool color = true);
```

```
    // Text functions (if you implement them)
    // You'll likely need to include a font array somewhere or have a separate font library.
    void setCursor(int x, int y);
    void printChar(char c, bool color = true);
    void printString(const char *str, bool color = true);
    void setTextSize(int size);

    // Update function: If you're using a framebuffer, this sends the buffer to the display.
    void display();

    // Low-level command/data write functions (often private, but can be public for advanced
users)
    void writeCommand(byte cmd);
    void writeData(byte data);

    // You might need a private helper for selecting the correct controller
    void selectController(int x_coord);


private:
    // Pin assignments
    int _d0, _d1, _d2, _d3, _d4, _d5, _d6, _d7; // Data pins
    int _rs, _rw, _en; // Register Select, Read/Write, Enable
    int _cs1, _cs2; // Chip Select for the two WS0010 controllers

    // Framebuffer (if you decide to use one, highly recommended for graphics)
    // A 128x64 display needs 128 * 64 / 8 = 1024 bytes (plus maybe extra for padding/dual
controller).
    // The WS0010 manages its own memory, but a software buffer makes drawing easier.
    byte _buffer[WS0010_WIDTH * WS0010_HEIGHT / 8]; // Example for a 1-bit monochrome
display

    // Private helper methods for low-level interaction or internal logic
    void _sendByte(byte data, bool is_command);
    void _setPinModes();
    void _setAllDataPins(byte value);
    // ... any other internal helper functions
};

#endif // WS0010_DISPLAY_H
```

Header File Essentials:

- **Include Guards (#ifndef/#define/#endif)**: Prevent multiple inclusions of the header

file, which can cause compilation errors.

- **#include <Arduino.h>**: Provides access to Arduino-specific functions and types (e.g., byte, pinMode, digitalWrite, delay).
- **Constants/Enums**: Define display dimensions, pin numbers (if fixed), or command codes.
- **Class Definition**:
  - **Constructor**: Takes parameters for the pins connected to your display. This is how the user "wires" the software to their hardware.
  - **public: methods**: These are the functions users will call to interact with your display (e.g., begin(), clearDisplay(), setPixel(), printString(), display()).
  - **private: members**: Internal variables (like pin assignments, the display buffer) and helper functions that shouldn't be directly accessed by the user.

3. Source File (WS0010.cpp - now potentially YourLibraryName.cpp)

This file contains the actual implementation of the methods declared in your header file.

```
#include "YourLibraryName.h" // Include your own header first
// You might need other includes, e.g., for specific fonts if you have them in separate files.
// #include "MyFont.h"

// Constructor implementation
WS0010_Display::WS0010_Display(int d0_pin, int d1_pin, int d2_pin, int d3_pin,
                               int d4_pin, int d5_pin, int d6_pin, int d7_pin,
                               int rs_pin, int rw_pin, int en_pin, int cs1_pin, int cs2_pin)
  : _d0(d0_pin), _d1(d1_pin), _d2(d2_pin), _d3(d3_pin),
    _d4(d4_pin), _d5(d5_pin), _d6(d6_pin), _d7(d7_pin),
    _rs(rs_pin), _rw(rw_pin), _en(en_pin),
    _cs1(cs1_pin), _cs2(cs2_pin) {
  // Constructor initializes private member variables with the provided pin numbers.
  // No hardware interaction here, just storing values.
}

// begin() method implementation
void WS0010_Display::begin() {
  // Set all control and data pins to OUTPUT mode
  pinMode(_d0, OUTPUT); // Repeat for all D1-D7, RS, RW, EN, CS1, CS2
  // ...
  pinMode(_cs1, OUTPUT);
  pinMode(_cs2, OUTPUT);
  pinMode(_rs, OUTPUT);
  pinMode(_rw, OUTPUT);
  pinMode(_en, OUTPUT);
```

```
    // De-select both controllers initially
    digitalWrite(_cs1, HIGH);
    digitalWrite(_cs2, HIGH);

    // Initial display reset sequence (if required by WS0010 datasheet)
    // For WS0010, you might need a brief delay and then issue a display ON command.
    delay(10); // Example delay
    // Send initial configuration commands to both controllers
    // You'll need to send commands to each controller separately if they manage different
halves.
    selectController(0); // Select first controller (e.g., left half)
    writeCommand(0x30); // Function Set: Basic instruction set
    writeCommand(0x30); // Function Set: Basic instruction set (repeated if needed)
    writeCommand(0x0C); // Display On
    // ... more initialization commands from the WS0010 datasheet
    selectController(WS0010_WIDTH / 2); // Select second controller (e.g., right half)
    writeCommand(0x30); // Function Set: Basic instruction set
    writeCommand(0x30); // Function Set: Basic instruction set
    writeCommand(0x0C); // Display On
    // ... more initialization commands

    clearDisplay(); // Clear any junk on startup
    display();      // Send the blank buffer to the display
}

// Private helper to set all data pins to a specific byte value
void WS0010_Display::_setAllDataPins(byte value) {
    digitalWrite(_d0, (value & 0x01) ? HIGH : LOW);
    digitalWrite(_d1, (value & 0x02) ? HIGH : LOW);
    digitalWrite(_d2, (value & 0x04) ? HIGH : LOW);
    digitalWrite(_d3, (value & 0x08) ? HIGH : LOW);
    digitalWrite(_d4, (value & 0x10) ? HIGH : LOW);
    digitalWrite(_d5, (value & 0x20) ? HIGH : LOW);
    digitalWrite(_d6, (value & 0x40) ? HIGH : LOW);
    digitalWrite(_d7, (value & 0x80) ? HIGH : LOW);
}

// Private helper to send a byte (command or data) to the display
void WS0010_Display::_sendByte(byte data, bool is_command) {
    digitalWrite(_rs, is_command ? LOW : HIGH); // RS low for command, high for data
    digitalWrite(_rw, LOW); // Write mode (RW low)
```

```
    _setAllDataPins(data); // Put data on the bus

    digitalWrite(_en, HIGH); // Enable pulse high
    delayMicroseconds(1);   // Small delay for enable pulse width
    digitalWrite(_en, LOW);  // Enable pulse low
    delayMicroseconds(1);   // Small delay for data setup
}

// writeCommand() method implementation
void WS0010_Display::writeCommand(byte cmd) {
    _sendByte(cmd, true);
}

// writeData() method implementation
void WS0010_Display::writeData(byte data) {
    _sendByte(data, false);
}

// selectController() method implementation
// This is crucial for dual-controller displays.
// Assuming controller 1 handles columns 0-63 and controller 2 handles columns 64-127.
void WS0010_Display::selectController(int x_coord) {
    digitalWrite(_cs1, HIGH); // De-select both initially
    digitalWrite(_cs2, HIGH);

    if (x_coord < WS0010_WIDTH / 2) {
        digitalWrite(_cs1, LOW); // Select controller 1
    } else {
        digitalWrite(_cs2, LOW); // Select controller 2
    }
    delayMicroseconds(1); // Small delay for chip select setup
}


// clearDisplay() method implementation
void WS0010_Display::clearDisplay() {
    memset(_buffer, 0, sizeof(_buffer)); // Clear the software buffer
    // You might also send commands to clear the display hardware directly if desired,
    // but typically `display()` will take care of sending the cleared buffer.
}

// setPixel() method implementation
void WS0010_Display::setPixel(int x, int y, bool color) {
```

```
    if (x < 0 || x >= WS0010_WIDTH || y < 0 || y >= WS0010_HEIGHT) {
        return; // Out of bounds
    }

    // Calculate byte and bit position in the buffer
    // WS0010 is often page-oriented (8 pixels per byte vertically)
    // Byte address: (column + (page * width))
    // Bit mask: 1 << (row % 8)

    int page = y / 8; // Each page is 8 pixels high
    int bit_offset = y % 8; // Which bit within the byte (0-7)

    // Calculate index into the buffer. This depends on memory mapping.
    // A common mapping: each byte holds 8 vertical pixels.
    // Index = (page * WIDTH) + column
    int bufferIndex = (page * WS0010_WIDTH) + x;

    if (color) {
        _buffer[bufferIndex] |= (1 << bit_offset); // Set bit
    } else {
        _buffer[bufferIndex] &= ~(1 << bit_offset); // Clear bit
    }
}

// display() method implementation
void WS0010_Display::display() {
    // This function iterates through your software buffer and sends the data
    // to the display controllers. Since you have two, you need to manage them.

    for (int controller_idx = 0; controller_idx < 2; controller_idx++) {
        int start_col = (controller_idx == 0) ? 0 : (WS0010_WIDTH / 2);
        int end_col = (controller_idx == 0) ? (WS0010_WIDTH / 2) : WS0010_WIDTH;

        selectController(start_col); // Select the current controller

        for (int page = 0; page < WS0010_HEIGHT / 8; page++) {
            writeCommand(SET_PAGE_ADDRESS | page); // Set page address
            writeCommand(0x10); // Set high column address (usually 0 for start)
            writeCommand(0x00); // Set low column address (usually 0 for start)

            for (int col = start_col; col < end_col; col++) {
                // Adjust buffer index for the current controller
                int buffer_col = col;
```

```
            if (controller_idx == 1) { // If second controller, adjust column index for buffer lookup
               buffer_col = col - (WS0010_WIDTH / 2);
            }
            int bufferIndex = (page * WS0010_WIDTH) + buffer_col;

            writeData(_buffer[bufferIndex]);
         }
      }
   }
   // De-select both controllers after update
   digitalWrite(_cs1, HIGH);
   digitalWrite(_cs2, HIGH);
}


// Implement other drawing functions (drawLine, drawRectangle, etc.)
// These will typically use setPixel() internally.
// Example for drawLine (simplified Bresenham's)
void WS0010_Display::drawLine(int x0, int y0, int x1, int y1, bool color) {
   // Basic Bresenham's line algorithm
   int dx = abs(x1 - x0);
   int dy = abs(y1 - y0);
   int sx = (x0 < x1) ? 1 : -1;
   int sy = (y0 < y1) ? 1 : -1;
   int err = dx - dy;

   while (true) {
      setPixel(x0, y0, color);
      if (x0 == x1 && y0 == y1) break;
      int e2 = 2 * err;
      if (e2 > -dy) { err -= dy; x0 += sx; }
      if (e2 < dx) { err += dx; y0 += sy; }
   }
}

// ... implement other functions similarly
```

Source File Essentials:

- **#include "YourLibraryName.h"**: Crucial to include your own header so the compiler knows the declarations for the methods you're implementing.
- **Method Implementations**: For each function declared in the .h file, you provide its full implementation here.
- **Low-Level Interaction**: This is where you'll have your digitalWrite() and pinMode() calls

for the specific WS0010 commands and data transfer.
- **Framebuffer (_buffer)**: For graphical displays, it's common to draw to an in-memory buffer first and then transfer the entire buffer to the display using a display() or update() function. This prevents flicker and makes drawing operations more efficient.
- **Controller Management**: With two WS0010 controllers, your selectController() and display() functions will be critical to addressing the correct half of the screen.

4. library.properties File

This file provides metadata about your library. It's essential for the Arduino Library Manager and for users. Create this file directly inside your main YourLibraryName/ directory.

```
name=WS0010_OLED_Driver
version=1.0.0
author=Your Name <your.email@example.com>
maintainer=Your Name <your.email@example.com>
sentence=A driver for graphic OLED displays using dual WS0010 controllers.
paragraph=This library provides a comprehensive set of functions to control OLED displays powered by two WS0010 chips in graphical mode. It includes basic drawing primitives like pixels, lines, rectangles, and text.
category=Display
url=https://github.com/yourusername/WS0010_OLED_Driver (if you plan to put it on GitHub)
architectures=avr,esp8266,esp32
```

Key Fields:

- name: The name of your library.
- version: Semantic versioning (e.g., 1.0.0, 1.0.1, 2.0.0).
- author, maintainer: Your contact information.
- sentence, paragraph: Short and long descriptions of your library.
- category: Helps users find your library (e.g., Display, Sensors, Communication).
- url: Link to your GitHub repository (if applicable).
- architectures: Specify which Arduino board architectures your library supports (e.g., avr for Uno, Nano; esp8266, esp32).

5. keywords.txt File

This file tells the Arduino IDE which words in your code should be highlighted. It helps users quickly identify functions and classes from your library. Create this file in the main YourLibraryName/ directory.

```
WS0010_Display    KEYWORD1
begin             KEYWORD2
clearDisplay      KEYWORD2
setPixel          KEYWORD2
drawLine          KEYWORD2
drawRectangle     KEYWORD2
```

display KEYWORD2
writeCommand KEYWORD2
writeData KEYWORD2
WS0010_WIDTH LITERAL1
WS0010_HEIGHT LITERAL1

Structure: Keyword TAB Type

- KEYWORD1: Typically for class names.
- KEYWORD2: For public methods.
- LITERAL1: For constants or enums.

6. examples/ Directory

This is incredibly important! Users often start by looking at example sketches.

```
YourLibraryName/
└── examples/
    └── BasicGraphics/
        └── BasicGraphics.ino
    └── TextDisplay/
        └── TextDisplay.ino
    └── AnimationDemo/
        └── AnimationDemo.ino
```

Example BasicGraphics.ino:

```
#include <WS0010_OLED_Driver.h> // Include your library header

// Define the pins connected to your display
// (Replace with your actual pin assignments)
#define D0_PIN 2
#define D1_PIN 3
#define D2_PIN 4
#define D3_PIN 5
#define D4_PIN 6
#define D5_PIN 7
#define D6_PIN 8
#define D7_PIN 9
#define RS_PIN 10
#define RW_PIN 11
#define EN_PIN 12
#define CS1_PIN 13
#define CS2_PIN A0

// Create an instance of your display driver object
```

```
WS0010_Display display(D0_PIN, D1_PIN, D2_PIN, D3_PIN, D4_PIN, D5_PIN, D6_PIN, D7_PIN,
                       RS_PIN, RW_PIN, EN_PIN, CS1_PIN, CS2_PIN);

void setup() {
  Serial.begin(9600);
  Serial.println("Initializing WS0010 OLED...");

  display.begin(); // Initialize the display hardware
  Serial.println("Display initialized.");

  display.clearDisplay(); // Clear the buffer
  display.display();      // Send empty buffer to display
  delay(1000);

  // Draw some pixels
  for (int i = 0; i < WS0010_WIDTH; i += 5) {
    display.setPixel(i, i / 2, true);
  }
  display.display();
  delay(1000);

  // Draw a line
  display.clearDisplay();
  display.drawLine(0, 0, WS0010_WIDTH - 1, WS0010_HEIGHT - 1, true);
  display.display();
  delay(1000);

  // Draw a rectangle
  display.clearDisplay();
  display.drawRectangle(10, 10, 50, 30, true, false); // Outline
  display.drawRectangle(70, 20, 40, 20, true, true);  // Filled
  display.display();
  delay(1000);

  // You can add text if you implement it
  // display.clearDisplay();
  // display.setCursor(0, 0);
  // display.printString("Hello OLED!", true);
  // display.display();
  // delay(2000);
}

void loop() {
```

```
  // You can add animations or dynamic content here
  // For this basic example, we just keep the last drawn content.
}
```

7. README.md (Highly Recommended)

A README.md file in the root of your library directory (alongside src, examples, etc.) provides essential information for users.

Content for README.md:

- **Library Name & Description:** What it does.
- **Features:** List the capabilities (e.g., "Pixel drawing," "Line drawing," "Supports 128x64 resolution," "Dual controller support").
- **Compatibility:** Which Arduino boards (Uno, Mega, ESP32, ESP8266 etc.) and IDE versions.
- **Installation:** How to install (usually through Arduino Library Manager or manually).
- **Wiring Guide:** Crucial for a display driver! Provide a clear table or diagram showing how to connect the WS0010 pins to Arduino pins.
- **Usage Examples:** Brief code snippets showing how to initialize and use basic functions (referencing the examples/ folder).
- **API Reference:** A list of public functions with brief descriptions.
- **Troubleshooting/FAQs:** Common issues and solutions.
- **Contributing:** How others can contribute to your library.
- **License:** State the license under which your library is distributed (e.g., MIT, GPL).

8. Key Driver Concepts and Best Practices

- **Framebuffer:** As mentioned, maintaining an in-memory buffer (_buffer) that represents the display's pixels is a standard and efficient approach for graphics. You draw to this buffer, and then call display() to send the entire buffer to the actual screen.
- **Low-Level vs. High-Level:** Separate the concerns. Your _sendByte, writeCommand, writeData functions handle the specific WS0010 electrical signaling. Your setPixel, drawLine, drawRectangle functions use these low-level functions to build more complex graphics.
- **Abstraction:** Hide the complexity of the dual WS0010 controllers from the user. They should just call setPixel(x,y) and your driver should internally figure out which controller (CS1 or CS2) to activate.
- **Modularity:** Break down complex tasks into smaller, manageable functions (e.g., separate functions for initializing, clearing, drawing shapes, sending data).
- **Error Handling:** For a driver, consider how to handle invalid inputs (e.g., setPixel with x or y out of bounds). You might just return, or print an error to Serial.
- **Efficiency:** For embedded systems, efficiency is key. Minimize digitalWrite() calls where possible (e.g., by setting multiple pins at once if your hardware allows), and optimize your drawing algorithms.

- **Comments:** Comment your code extensively, explaining the purpose of functions, complex logic, and any hardware-specific considerations.

By following this structure, you'll not only have a functional driver but also one that is well-documented, easy for others to use, and maintainable! Good luck!