LED Strips Sign Board Documentation

by:

Ahmad Azeez (Website, Web Server, & Web Server-to-Board Communication)

Wezi Kaonga (Remote Communication & Display)

Jay Mistry (Bluetooth Communication & Webserver-to-Board Communication)

Ripanjot Singh (Font Design & Implementation)

Gurchandan Singh (Display Backend & Character Rendering)

Overview

The LED Strips Sign Board is a web and remote-controlled sign board powered by an Arduino Mega. It can display **text**, **time**, and **custom art**, with a web interface for easy control and customization.

This documentation contains everything that needs to be known about the inner machinations of the **Arduino, web serve**r, and **website**.

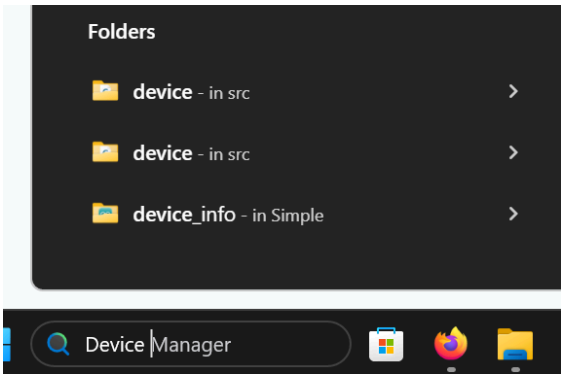
Table of Contents

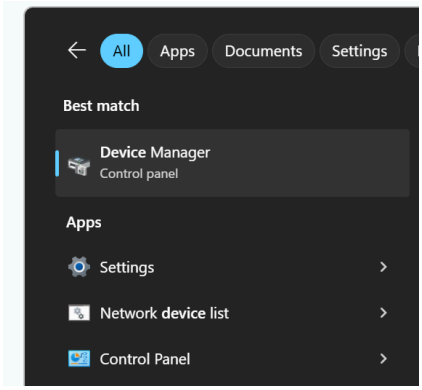
How To Use

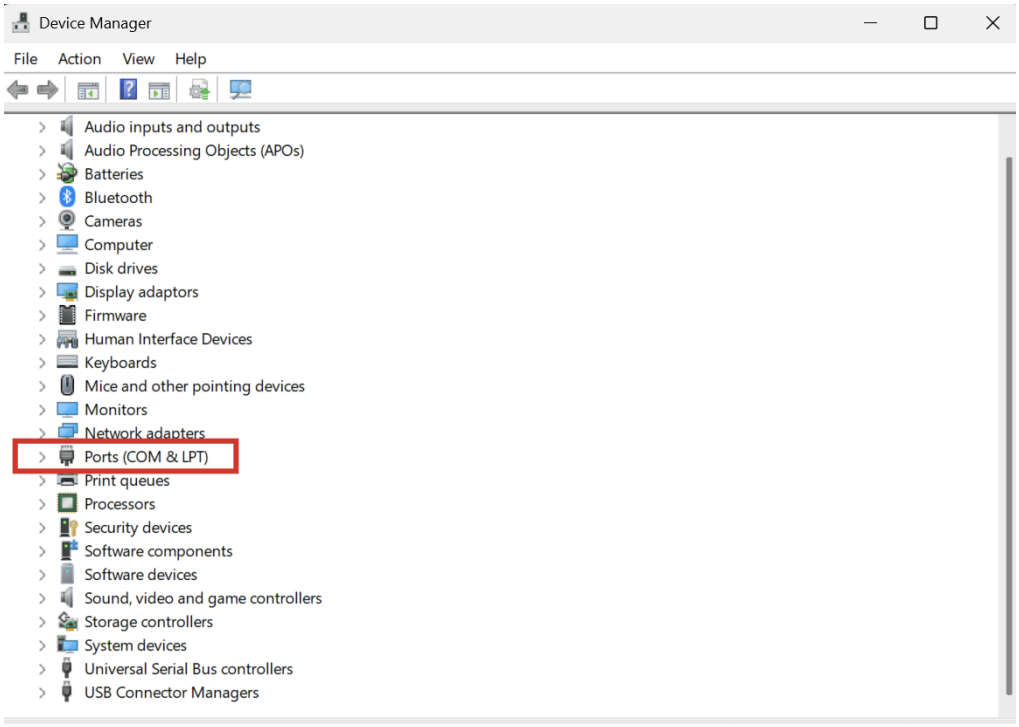
Wired Connection

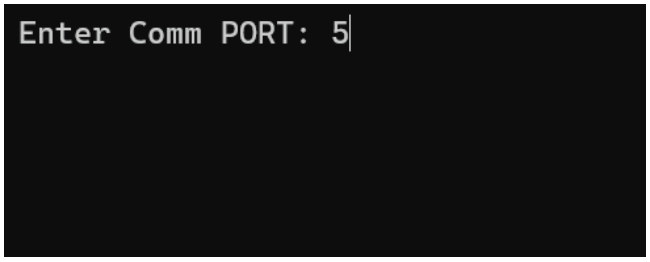
1. Power on the board.
2. Connect the board to your computer using the provided **USB-B to USB-A** cable.

3. Navigate to the **"Website"** folder and double-click **"start.bat"** to launch the web server.
4. When prompted, enter the **COM** port number that the board is connected to (this can be found in your **“Device Manager”**.

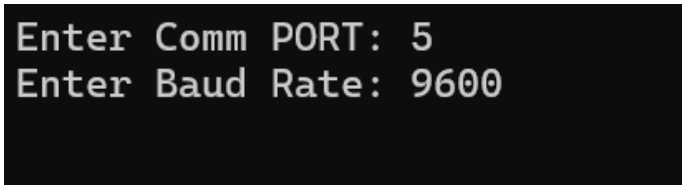




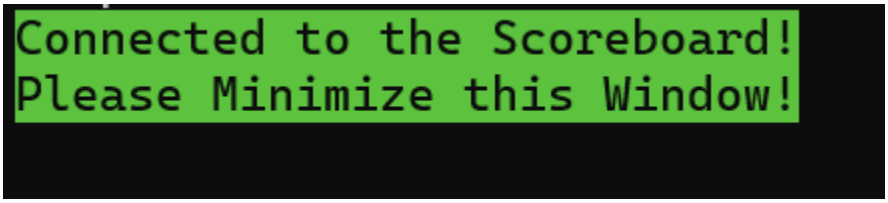




5. Enter the baud rate (**9600** recommended).

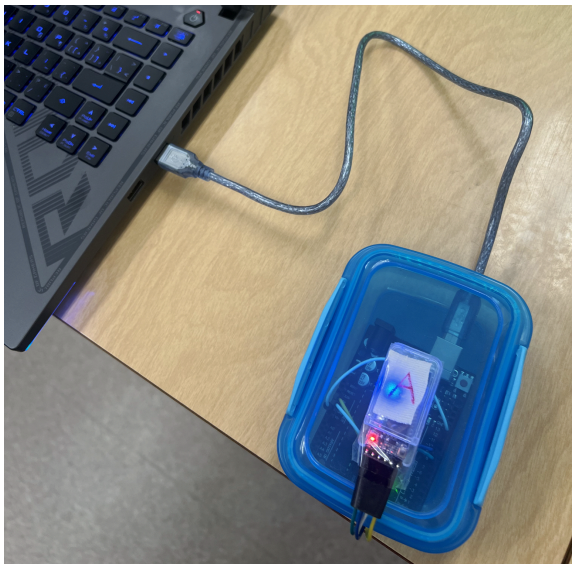


6. The website should now be fully connected to the board.

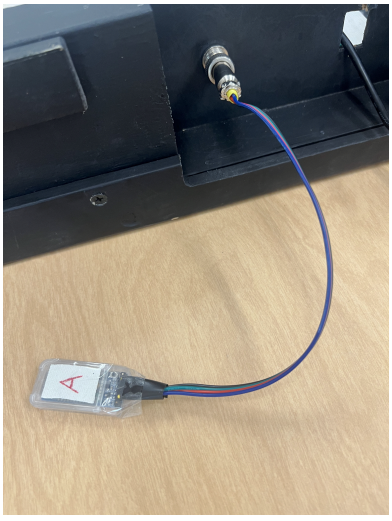


Wireless Connection

1. Power on the board.
2. Connect the **sending** bluetooth device (the blue box) to the computer.



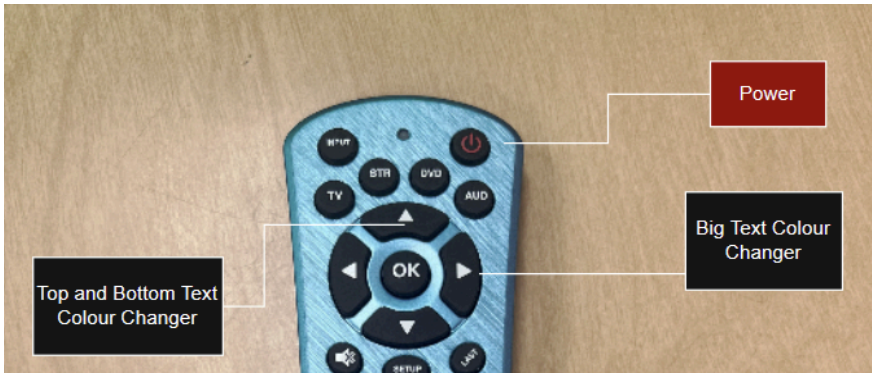
3. Connect the receiving bluetooth device to the board.



4. You will notice that the **2** bluetooth devices are blinking **rapidly**. **Wait** until they connect. When they are connected, they would both start blinking at a much **slower** pace.
5. Go to the "Website" folder and double-click "start.bat" to launch the web server.
6. When prompted, enter the COM port number that the board is connected to.
7. Enter the baud rate (9600 recommended).
8. The website should now be wirelessly connected to the board.

With a Remote

1. Power on the board.
2. Press the **"Power"** button on the remote to activate it for the board you want to control.



3. The board should now be ready to receive commands from the remote.

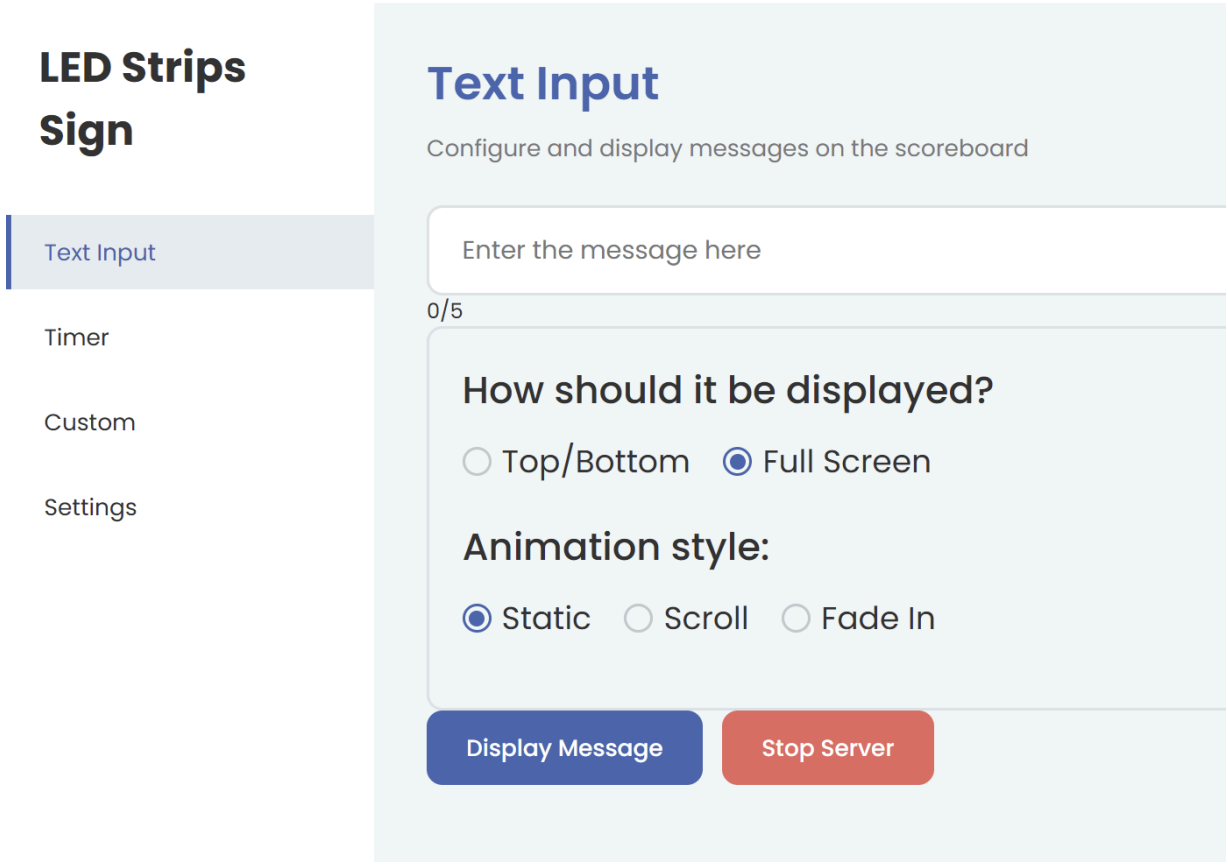
IMPORTANT NOTES:

- If it seems like the remote isn't working, press the **"TV"** button and try again.
- The cable and Bluetooth share the same port, so they cannot be used simultaneously.
- If the board crashes, first shut down the server. Then navigate to **Board** > **App** and double-click on **"ArduinoResetter.exe"**. Click **"Reset Arduino"** to restart the board without needing to power it **off** and **on**.

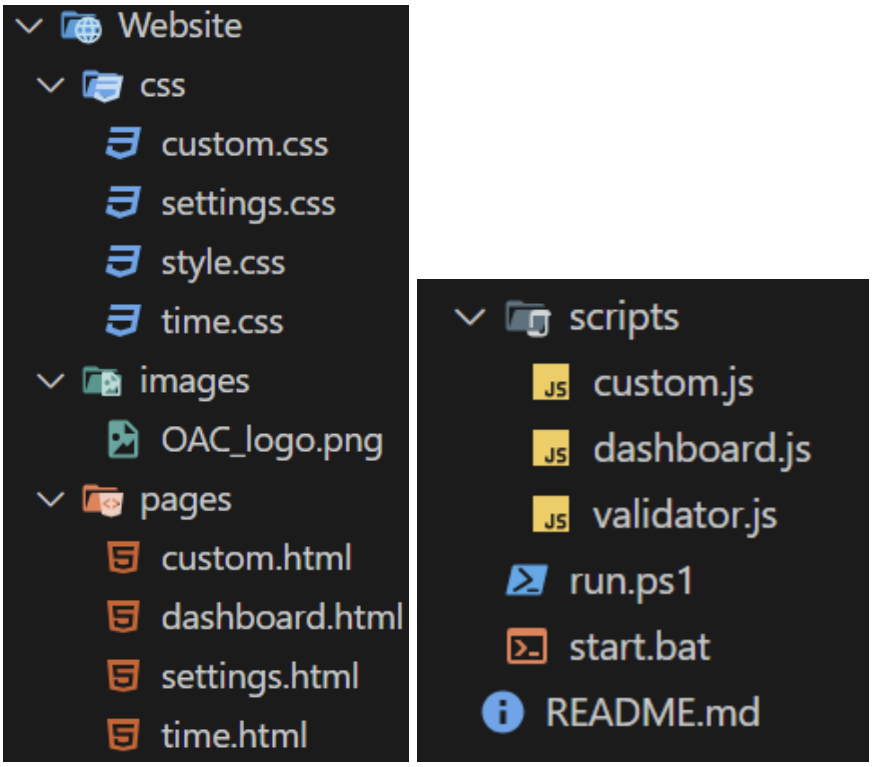
Website

This part of the documentation will contain everything that needs to known about the website. The **frontend**, **backend** and **web server.**

Frontend



File Structure



- The **“css”** folder contains all the **css** files
- **“Images”** contains all the **images** used
- **“pages”** contains all the website pages
- **“scripts”** contains all the **javascript** scripts used
- **“start.bat”:** Runs the **“run.ps1”** powershell script
- **“run.ps1”:** Starts a web server

“start.bat”, “run.ps1”, and all the javascripts are explained in detail in the **“Backend”** section.

The codebase contains **4** directories: **pages**(all html files),c(all stylesheets), **scripts**(javascript files), **images**(images and logos).

Dashboard.html

This page has validator.js, style.css and Dashboard.js linked.

Validator.js will change the layout and css depending on what radio buttons are changed.

When the **Display Message** button is clicked, it will call the **sendMessage()** which will do the following:

- Perform pre checks of input and selected options
- A post request will be made using fetch to endpoint **/dashboard/post** with json data consisting of keys: **command, isBig, data.** The button will stay disabled and after 4 seconds it will be enabled again.

When the Stop **Server** button is clicked, **killServer()** is called which will make a get request using fetch to endpoint **/kill** and if successful, it will alert the user on the webpage.

Time.html

This file has time.css, style.css and Dashboard.js linked.

When any of the **preset timer buttons** are clicked, it will replace the content of the timer input automatically via javascript.

When the **Start button** is clicked, **start_timer()** is called which will make a post to **/dashboard/post** with the json data having keys command, isBig and data. Command will be just **sTimer**, isBig will be yes always, data will be **min,sec**.
[Button will be disabled for 4 seconds, then re enabled using **setTimeout()**]

When the **Pause button** is clicked, **pause_and_resume_timer()** is called which will first change its text content and background color, than make a post to **/dashboard/post** having keys **command** which will be either **pTimer** or **resume** depending on the textcontent of the button when pressed.
[Button will be disabled for 4 seconds, then re enabled using **setTimeout()**]

When the **Reset button** is clicked, **reset_timer()** is called which will make a post to **/dashboard/post** with keys **command, isBig and data**. Command will be just **sTimer**, isBig will be yes always, data will be **min,sec**.
[Button will be disabled for 4 seconds, then re enabled using **setTimeout()**]

Time of day button, when pressed will make a post to **dashboard/post** with **command** as **tod**

Settings.html

Send Settings button will select the color hex values of **top, bottom and full-screen** along with the **brightness level** you can select via the slider, upon clicking a call will be made to **send_settings()** method that will post keys: **command, brightness, tcolor, bcolor, fcolor** to the **/dashboard/post** [Button will be disabled for 4 seconds, then re enabled using **setTimeout()**]

Custom.html

This provides a digital board and buttons to send custom drawings to the board, clear all and erase. There are multiple javascript functions in **custom.js** that keep track of and update the background-color of digital pixels on the page as the user draws.

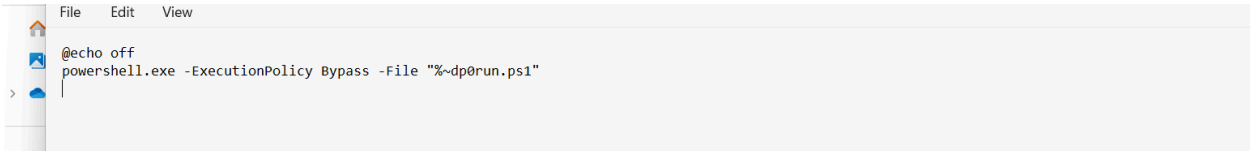
A set data structure is used to store the drawing in pair format: **[row,col,color_hex]** . When **Send Display** is clicked, the set is divided into chunks of 15 pairs and then gets posted to **/dashboard/post** every 2 seconds. For every button press the first batch will have the param as **start** and the rest will have **no**, this is to signal the board to clear the buffer and start a new drawing.

Backend

The backend codebase consists of a lightweight powershell server listening to receive commands from the signboard website via get and post requests. The webpage code to enable custom and seamless communication with the scoreboard. The wireless bluetooth IOT code in order to push the data from the local system to the main scoreboard.

Powershell script (Run.ps1)

The run.ps1 will be started by calling the start.bat file that will bypass any execution policies and than run the script in current directory using powershell



Run.ps1 starts by checking if the port 8080 is currently not in use by calling **Get-NetTCPConnection**, if so then it will use it as the default otherwise keep asking the user to enter a different port for the server using the while loop.

```
$http = [System.Net.HttpListener]::new();

$server_port = 8080
if(Get-NetTCPConnection -LocalPort $server_port -ErrorAction SilentlyContinue){
    do{
    $server_port = Read-Host "Enter Server Port"
    if ([string]::IsNullOrWhiteSpace($server_port) -or $server_port -match '\D') {
        Write-Host "Enter a valid port"
    }
    elseif ([int]::Parse($server_port) -ge 1 -and [int]::Parse($server_port) -le 65535){

        if(Get-NetTCPConnection -LocalPort $server_port -ErrorAction SilentlyContinue){
            Write-Host "Port is already in use, enter a different one"
        }
        else{
            break
        }
    }
    else{
        Write-Host "Invalid Port"
    }
}
while($true)
}
```

It will ask for the COM serial port to use (this will differ from system to system), baud rate.

It will start the http server and than open the port as follows:

```
$http.Prefixes.Add($url);
$http.Start();

$out_port_a = new-Object System.IO.Ports.SerialPort $signPort,$baudRate,None,8,one
$out_port_a.open()
```

Here $singport is the serial port
None means no parity
8 bits represent a single character
One is the stopping bit

The logic to prompt for server listening or not is as follows:

```
if ($http.IsListening)
{
    write-host "Connected to the Scoreboard!"
    write-host "Please Minimize this Window!"
}
```

```
function send_display($data)

{
    # Uncomment this line when ready to use actual serial port
    $out_port_a.WriteLine($data)
    write-host($data)
}

function pad-text($value)
{

    if($value.command -eq "pTimer"){
        return  "$" + $value.command + "$"
    }
    elseif($value.command -eq "rTimer"){
        return  "$" + $value.command  + "$"
    }
    elseif($value.command -eq "settns"){
        return "$" + $value.command + "$" +  "[" + $value.brightness + ", " +  $value.tcolor + ", " +  $value.bcolor + ", " + $value.fcolor + "]"
    }
    elseif($value.command -eq "custom"){
        return "$" + $value.command + "$" + $value.param + "[" + $value.data + "]"
    }
    else{
        return "$" + $value.command + "$" + $value.isBig + "[" + $value.data + "]"
    }

}
```

Send_display method will just send the data to board via **serial_port.writeline**. It will additionally write it to the powershell console in order to log the data received.

Pad-text function is to construct the message as per the communication protocol format: '$command$displayStyle$[MESSAGE]' . Other variations are explained in the Communication section.

In order to prevent CORS issues that may arise due to system or browser settings, the following function in the script will add CORS headers for each request processed by the server as shown below:

```
# Function to add CORS headers to response
function Add-CorsHeaders($response)
{
    $response.AppendHeader("Access-Control-Allow-Origin", "*")
    $response.AppendHeader("Access-Control-Allow-Methods", "GET, POST, OPTIONS")
    $response.AppendHeader("Access-Control-Allow-Headers", "Content-Type, Accept, X-Requested-With")
    $response.AppendHeader("Access-Control-Max-Age", "86400")
}
```

The Append-Header method inside the custom Add-CorsHeaders function will be used to allow all origins, accept get, post requests and other two are just for content-type and request age.

The while loop will keep listening for incoming requests, extract the api path and than accordingly take the action, we are using the following two paths:

/dashboard/post -> This route will check for incoming post requests with the commands and data from the webpage and will utilize the pad-text method.

/kill -> When a get request is made to this route, it will kill the powershell server.

```
if ($context.Request.HttpMethod -eq 'POST' -and $context.Request.RawUrl -eq '/dashboard/post') {
    $FormContent = [System.IO.StreamReader]::new($context.Request.InputStream).ReadToEnd()
    $dataToSend = $FormContent | ConvertFrom-Json

    if ($dataToSend.command -eq 'postRaw') {
        send_display($dataToSend.data)
    }
    else {
        $formattedData = pad-text($dataToSend)
        send_display($formattedData)
    }

    Add-CorsHeaders -response $context.Response
    $context.Response.StatusCode = 200
    $buffer = [System.Text.Encoding]::UTF8.GetBytes("Success")
    $context.Response.ContentLength64 = $buffer.Length
    $context.Response.OutputStream.Write($buffer, 0, $buffer.Length)
    $context.Response.OutputStream.Close()
    continue
}
```

```
# Handle GET requests
if ($context.Request.HttpMethod -eq 'GET' -and $context.Request.RawUrl -eq '/kill') {
    Write-Host "Stopping the Server"
    Add-CorsHeaders -response $context.Response
    $context.Response.StatusCode = 200
    $buffer = [System.Text.Encoding]::UTF8.GetBytes("Server stopped")
    $context.Response.ContentLength64 = $buffer.Length
    $context.Response.OutputStream.Write($buffer, 0, $buffer.Length)
    $context.Response.OutputStream.Close()
    $http.Stop()
    break
}
```

The Add-Cors method will be used for all requests and responses.

Communication

This part of the documentation contains everything you need to know about the communication protocol and format.

Format

All messages follow this format: **$command$param[values]**
[ ] - The square brackets will be used to indicate the start and end of message

Command list:

- **sTimer** - will get transformed to $sTimer$, this is to start the countdown timer on board
- **rTimer**: Will transform to $rTimer$, this is to reset the started timer
- **pTimer**: Will become $pTimer$ used to pause the timer
- **Resume**: Will become $resume$, to resume the paused timer
- **settns** : Will transform to $settns$[brightness_value, #top_color_value, #bottom_color_value, #full-screen-color], this is to update the display settings
- **custom:** Will transform to either $custom$start$[(row,col,#color_val)] or $custom$no$[(row,col,#color_val)], this is to show the custom drawing from the digital board.

Text Message Format: This will be used for displaying normal font messages like static, scroll, etc. The format: **$command$display_style$[messages]**

Bluetooth

The codebase for wireless bluetooth will be running on an Arduino uno chip. It will have two main functions: setup() and loop().

Setup will initialize the pins rx and tx as 9 and 8 respectively.

Loop will constantly receive the data from the powershell server using the Serial.available method, read it using the .read() and then send it to arduino mega using the .write() method.

Remote

For the board to communicate with any remote, the board is installed with an IRreceiver to send and receive Infrared signals between the board and the remote. This allows the user to send quick preset commands that the user might want displayed onto the board.

Arduino Mega/Scoreboard

This part provides a detailed explanation of the arduino mega (the scoreboard itself) code.

MainBoard.ino

MainBoard.ino manages the **display**, **timers**, **remote control input**, and processes **commands** sent via serial communication.

Libraries

```
#include "Display.h"
#include "Timer.h"
#include <Wire.h>
#include <RTClib.h>
#include "Remote.h"
```

- **Display.h** - Handles display-related operations (displaying text and custom art).
- **Timer.h** - Manages countdown and scheduling timers.
- **Wire.h** - Enables I2C communication (used by RTC module).
- **RTClib.h** - Library to interface with the DS3231 RTC module.
- **Remote.h** - Handles IR remote control operations.

Global Variables

```
RTC_DS3231 rtc;
bool useBigFont = true;
Timer timer;
RemoteControl remote;
```

- **rtc** is used to interact with the real-time clock.
- **useBigFont** determines the font size of displayed text.
- **timer** is responsible for countdown and scheduling.
- **remote** is responsible for decoding remote control signals.

Serial Communication Variables

```
int messageSize = 0;
```

command stores parsed serial input commands.

isBig indicates whether to use the **big** or **small** font for display.
message and **message2** store the primary and secondary text to display.
messageSize tracks the length of input messages.

Raw Input Handling

```
bool dataToSend = false;
```

- **numRawChar** defines a maximum character limit for raw serial input.
- **msgRaw** stores raw input data.
- **charCount** counts received characters.
- **dataToSend** is set when a complete message is ready for processing.

Display Singleton Instance

```
bool dataToSend = false;
Display& display = Display::getInstance();
```

Ensures only one Display instance exists using a singleton pattern.

Setup Function

```
void setup()
```

Initialization

```
display.setup(4);
```

- Initializes serial communication at **9600** baud.
- Sets up **RTC** and remote control.
- Initializes the display, using brightness level 4.

Display Initial Message

```
display.displayText("LED Strips", "Sign", "static", "no");
```

Shows the initial screen with text: **“LED Strips”** and **“Sign”**.

Loop Function

```
void loop()
```

Serial Input Processing

```
if (Serial.available())
{
      String input = Serial.readStringUntil('\n');
      parseInput(input);
}
```

Checks for incoming serial data and processes it using **parseInput()**.

Remote Control Handling

```
remote.useRemote();
```

Handles **IR** remote input.

Timer Update

```
timer.updateTimer();
```

Continuously updates the timer state.

“parseInput” Function

```
void parseInput(String input)
```

Parses commands received from the serial monitor.

Commands follow the format: **$command$param[values]**.

Validation

```
if (input.charAt(0) != '$')
```

Ensures valid commands start with **$**.

Extracting the Command

```
command = input.substring(firstDollar + 1, secondDollar);
```

Extracts the main command between **$** symbols.

Handling Different Commands

```
if (command == "custom")
```

Calls **display.displayCustomPixels()** for custom patterns.

```
else if (command == "settns")
```

Calls **updateSettings()** to modify settings.

```
else if (command == "sTimer")
```

Calls **timer.parseTimerInput()** to start a countdown timer.

```
else if (command == "pTimer")
```

Calls **timer.pauseTimer()** to pause the timer.

```
else if (command == "rTimer")
```

Calls **timer.resetTimer()** to reset the timer.

```
else if (command == "resume")
```

Calls **timer.resumeTimer()** to continue a paused timer.

```
else
```

Treats any other input as a regular message for display.

“showMessage” Function

```
void showMessage(String input)
```

Extracts and displays messages from input.

```
if (isBig == "no")
```

Extracts **message** and **message2** when using small font.

```
char currentCommand[command.length() + 1];
command.toCharArray(currentCommand, command.length() + 1);
```

Converts **String** variables to **char[]** for **displayText()** function.

```
display.displayText(currentMessage, currentMessage2, currentCommand,
currentIsBig);
```

Displays the extracted text.

“updateSettings” Function

```
void updateSettings(String input)
```

Extracting Values

```
int openBracket = input.indexOf('[');
int closeBracket = input.indexOf(']');
```

Finds the range of setting parameters.

```
int firstComma = input.indexOf(',', openBracket);
int secondComma = input.indexOf(',', firstComma + 1);
int thirdComma = input.indexOf(',', secondComma + 1);
```

Finds comma positions for splitting values.

Brightness Parsing

```
String brightnessStr = input.substring(openBracket + 1, firstComma);
int brightness = brightnessStr.toInt();
display.setBrightness(brightness);
```

Extracts brightness value and updates display.

Colour Parsing

```
uint32_t topColr = (uint32_t)strtoul(topColour.c_str() + 1, NULL,
16);
uint32_t bottomColr = (uint32_t)strtoul(bottomColour.c_str() + 1,
NULL, 16);
uint32_t fullColr = (uint32_t)strtoul(fullColour.c_str() + 1, NULL,
16);
```

Converts colour hex values to integer format.

Updating Colours

```
display.setTopColour(topColor);
display.setBottomColour(bottomColor);
display.setFullColour(fullColor);
```

Updates display colours.

```
display.displayText("Done", "", "static", "yes");
```

Displays Done to indicate successful update.

Display

Display.h

The Display.h file defines the Display class, which serves as the core display engine for rendering characters, text animations, and visual effects on the LED matrix strips. It follows the Singleton Pattern to ensure only one instance of the display controller exists.

LED Strip Management: Controls 15 horizontal LED strips using the Adafruit NeoPixel library.

Text Rendering: Provides support for different fonts.

Scrolling and Animation Effects: Supports multiple modes like SCROLL, FADE_IN, SCROLL_STOP, STATIC, STROBE, MARQUEE, etc.

Colour Management: Allows control over top, bottom, and full-text colours.

Custom Pixel Display: Supports manual drawing of LED pixels from a string format (e.g., `(x, y, color)`).

Fixed Width Rendering for Timer Display: Ensures visually consistent alignment in time-based formats like `MM:SS`.

Structure Breakdown

Singleton Setup

```
static Display* instance;
static Display& getInstance();
```

- Ensures only one `Display` object is active at runtime.

LED Configuration

```
const int stripPins[NUM_STRIPS];
Adafruit_NeoPixel strips[NUM_STRIPS];
uint32_t frameBuffer[NUM_STRIPS][NUMPIXELS];
```

- Defines pins for each LED strip and a framebuffer to hold current pixel states.

Drawing Functions

```
int getCharacterWidth7x7(char c);
int getCharacterWidth15x15(char c);
void drawCharacter7x7(char c, int x, int y, uint32_t color);
void drawCharacter15x15(char c, int x, int y, uint32_t color);
```

- Used to draw individual characters using two font types with dynamic width measurement for spacing consistency.

Display Effects

```
void scrollTextContinuous(...);
void scrollTextAndStop(...);
void fadeInText(...);
void displayStaticText(...);
```

- Encapsulates different modes of text animations. These are called by the public `displayText()` function.

Public Interface

```
void setup(int brightness);
void updateLEDs();
void setPixel(...);
```

```
void clearBuffer(...);
void displayText(...);
```

- `setup()` initializes the strips.
- `displayText()` is the master controller that decides what animation and font rendering logic to use based on mode.

Colour Functions

```
void setTopColour(...);
void setBottomColour(...);
void setFullColour(...);
```

- Used for setting colour schemes for top text, bottom text, or entire display uniformly.

Custom Drawing

```
void displayCustomPixels(String input, String chunkPos);
```

- Allows for drawing a custom pattern from encoded string input in `(x,y,color)` format.

Display.cpp

The **`Display.cpp`** file implements the logic defined in **`Display.h`** and provides the full functionality needed to render text, handle effects, and manage pixel-level control of the LED strip matrix.

Initialization and Singleton Pattern

```
Display* Display::instance = nullptr;

Display& Display::getInstance() {
    if (!instance)
        instance = new Display();
    return *instance;
}
```

- Uses a **Singleton Design Pattern** to ensure there is only one global instance of the Display class.

```
Display::Display() {
    for (int i = 0; i < NUM_STRIPS; i++) {
        strips[i] = Adafruit_NeoPixel(NUMPIXELS, stripPins[i],
NEO_GRB + NEO_KHZ800);
    }
    clearBuffer(true);
}
```

- The constructor initializes all 15 NeoPixel strips and clears the display buffer.

```
void Display::setup(int brightness) {
    for (int i = 0; i < NUM_STRIPS; i++) {
        strips[i].begin();
        strips[i].setBrightness(brightness);
        strips[i].show();
    }
}
```

- Initializes LED strips with the given brightness level.

Frame Buffer Management

```
void Display::clearBuffer(bool bigFont);
void Display::updateLEDs();
```

```
void Display::setPixel(int x, int y, uint32_t color);
```

- **`clearBuffer()`** clears the display matrix (used before drawing new frames).
- **`updateLEDs()`** pushes the frame buffer content to the physical LEDs.
- **`setPixel()`** updates the buffer with a color for a specific x/y coordinate.

Character Rendering

```
int Display::getCharacterWidth7x7(char c);
int Display::getCharacterWidth15x15(char c);
```

- Calculates the actual width of a character by detecting pixel boundaries. Used for **dynamic character spacing**.

```
void Display::drawCharacter7x7(char c, int x, int y, uint32_t color);
void Display::drawCharacter15x15(char c, int x, int y, uint32_t
color);
```

- Draws individual characters on the frame buffer using font data.
- For big fonts, **15x15** font is used.
- Each character is auto-trimmed and drawn pixel by pixel.

Text Animation Modes

The main animation logic is implemented in:

```
void Display::scrollTextContinuous(...);
void Display::scrollTextAndStop(...);
void Display::fadeInText(...);
void Display::displayStaticText(...);
void Display::breatheText(...);
```

- **`scrollTextContinuous`**: Scrolls text from right to left in a loop.

- **`scrollTextAndStop`**: Scrolls once and then stops.
- **`fadeInText`**: Increases brightness gradually to simulate fade-in.
- **`displayStaticText`**: Centers the text on screen and displays it without animation.
- **breatheText**: Loops through the brightness function to give off a “Fade-in, fade-out” effect

These functions are called internally by the master function:

```
void Display::displayText(const char* text1, const char* text2, const char* command, const char* displayType);
```

- Acts as the entry point for all display behavior.
- **`command`** specifies the mode (e.g., **`"static"`**, **`"scroll"`**, **`"fade_in"`**).
- **`displayType`** tells whether to use big font or not (**`"yes"`** or **`"no"`**).

Colour Functions

```
void Display::setTopColour(uint32_t colourHex);
void Display::setBottomColour(uint32_t colourHex);
void Display::setFullColour(uint32_t colourHex);
```

- These functions allow dynamic setting of text colors.
- Used in remote and web controls to customize visuals.

Custom Pixel Art

```
void Display::displayCustomPixels(String input, String chunkPos);
```

- Supports drawing custom patterns like logos or designs based on pixel coordinates passed as a string.
- Input format: **`[(x,y,color), (x,y,color), ...]`**

Smart Design Decisions

- **Fixed Width Timer Box**: For `MM:SS` style timers, each character is padded to fit in a fixed-width box. This keeps the timer display perfectly aligned.
- **Colon Character Special Handling**: To improve visual spacing around `:`, custom logic sets its width to 4 pixels with asymmetric spacing.

Timer

The timer handles all the time related functionalities. The **“Timer”** class provides functionality for managing a countdown timer using the **DS3231 Real-Time Clock (RTC)**. It supports starting, pausing, resuming, stopping, and resetting the timer.

Singleton Setup

```
static Timer* instance;
static Timer& getInstance();
```

- Same as display, this singleton ensures there is one instance running throughout the entire board.

Functionalities

- Initializes and configures the **RTC**.
- Starts a **countdown timer** with a specified duration.
- **Pauses** and **resumes** the timer without resetting progress.
- **Stops** or **resets** the timer as needed.
- Checks if the timer is **running** or **paused**.
- Updates the timer state based on elapsed time.
- Parses and processes timer-related input commands.

Public Methods

- **Timer()**: Constructor to initialize the timer.
- **void setupRTC()**: Sets up the RTC module.
- **void startTimer(int minutes, int seconds)**: Starts the countdown for a given time.
- **void pauseTimer():** Pauses the active timer.
- **void resumeTimer()**: Resumes a paused timer.

- **void stopTimer()**: Stops the timer completely.
- **void resetTimer():** Resets the timer to its initial state.
- **bool getTimerRunning()**: Returns true if the timer is currently running.
- **bool getTimerPaused()**: Returns true if the timer is paused.
- **void updateTimer()**: Updates the timer state based on real-time progress.
- **void updateCountdown()**: Updates the countdown timer
- **void updateTimeOfDay()**: Updates current time of day
- **void parseTimerInput(String input)**: Parses external input commands to control the timer.

Private Variables

- **RTC_DS3231 rtc**: RTC instance for timekeeping.
- **DateTime targetTime**: Stores the target end time of the countdown.
- **TimeSpan remainingTime**: Holds the remaining time when paused.
- **bool timerActive**: Tracks whether the timer is currently running.
- **bool timerPaused**: Tracks whether the timer is paused.
- **unsigned long lastUpdateMillis**: Stores the last update timestamp for tracking elapsed time.
- **int currentMin**, **currentSec:** Stores the current countdown values.
- **TimerMode currentMode:** Switches between the current time and the countdown

IMPORTANT NOTE:

To display the time in this format **“00:00”,** it would have to inputted like this:

```
display.displayText("5+:40", "", "static", "yes");
```

“5+:40” not **“5:40”**.

Remote

This part of the documentation contains everything you need to know about the remote class.

Remote.h

Remote variables:

```
uint8_t bright;
```

```
bool remoteStatus;
unsigned int enteredValue;
int minu;
int tbIndex;
int fIndex;
bool timerInputMode;
String timerCodes[10] = {"fb04", "fa05", "f906", "f708", "f609",
"f50a", "f30c", "f20d", "f10e", "ee11"};
```

Bright: This stores the value for the brightness to be used each time the board turns on
remoteStatus: a bool value that tells the board whether the remote is on or off.
enteredValue: stores each digit entered from the custom timer.
Minu: stores the minutes. (Min cannot be used as it is a reserved word).
tbIndex: stores the current color combination value in terms of integers to be used in to cycle through the colors for top and bottom text.
fIndex: stores the current color combination value in terms of integers to be used to cycle through the colors for full text.
timerInputMode: a bool value to tell the board whether it's in the custom timer input or not.
timerCodes: an array to store the remote codes for each of the numbers from 0 to 9.

Remote.cpp

Libraries

```
#include "Display.h"
#include <IRremote.h>
#include "Remote.h"
#include "Timer.h"
```

Display.h: allows for use of functions from the Display class.
IRremote.h: allows for communication between remote and board.
Remote.h: allows for access to the Remote.h variables and functions.
Timer.h: handles access to the timer class for remote timer functions.

Functionality

SetupRemote()
Sets up the IRreceiver to be able to communicate with the remote and the functionality in the Remote class

adjustBrightness(float change)
Adjusts the brightness based on the input from the remote. Displays whether the brightness going up or down on the board and whether it has reached max brightness or minimum brightness

toggleRemote(String remoteCode)

Gets the power button hex code from the remote and switches the remote functionality on and off.

handleTimerCodes(String remoteCode)
Uses the timerCodes array to set the timer up with whichever numbers were pressed on the remote. This function also allows for the user to stop, start, pause and resume the timer.

setDefaultMessage(String remoteCode)
Gets the home button hex code from the remote and displays a preset message onto the board.

useRemote()
This allows for the remote to be used on the board by passing all the functionality to the main.ino class. This includes only allowing the remote functionality to work if remoteStatus is true and allowing the user to display the Time of Day.

getBrightness()
This returns the value stored in the bright variable to be used outside of the remote class.

changeFColourScheme()

This uses the fIndex variable to cycle between 3 different colors for the board.

changeTBColourScheme()

This function uses the tbIndex variable to cycle between 6 preset color combinations for the top and bottom text on the board.

manualTimerInput()

This function allows for a custom timer to be set using the number codes from the remote.

getNumberFromIR(String command)

A function that gets the specified number from the remote.

Character Sets

In order to display custom characters on the LED matrix, we built two character sets: a 7x7 and a 15x15 pixel representation of characters, Both stored on the program memory section to save some RAM for the main functions using PROGMEM. These are stored as 2D arrays of binary values (1 = LED ON, 0 = LED OFF). Each character is pre-defined in a lookup array and accessed by its index.

```
#include <avr/pgmspace.h>  // Required for PROGMEM
```

7x7 Character Set-

The 7x7 set is declared in CharacterSet.h as a 3D array of type bool:

```
const bool charSet7x7[76][7][7] PROGMEM = { ... };
```

- 76 represents the number of characters supported and thus can be easily changed based on font..

- The 7x7 grid stores binary pixel values per character(we tried both binary and hex).

- Although each character is placed in a 7x7 matrix, the actual width and height vary for each character depending on its design (e.g., 'I' is narrower than 'M').

like so:

```
// Letter 'A'
{
    {0,0,1,1,0,0},
    {0,1,0,0,1,0},
    {1,0,0,0,0,1},
    {1,1,1,1,1,1},
    {1,0,0,0,0,1},
    {1,0,0,0,0,1},
    {1,0,0,0,0,1}
}
```

A helper function (getCharIndex) maps input characters to their index in the array. This ensures that character lookup remains consistent—even if we modify the character designs—so long as we keep the indices unchanged:

```
int getCharIndex(char c) {
    if (c >= 'A' && c <= 'Z') return c - 'A';        // Uppercase
    if (c >= 'a' && c <= 'z') return c - 'a' + 26;   // Lowercase
    if (c >= '0' && c <= '9') return c - '0' + 52;   // Digits

    switch (c) {
        case '.': return 62;
        case ',': return 63;
        case ':': return 64;
        case ';': return 65;
        case '!': return 66;
        case '?': return 67;
        case '@': return 68;
        case '#': return 69;
        case '$': return 70;
        case '%': return 71;
        case '&': return 72;
        case '*': return 73;
        case '-': return 74;
        case '+': return 75;
        default: return -1; // Unsupported character
```

15x15 Character Set-

For higher-resolution text, we designed a custom 15x15 font in CharacterSet15x15Hex.cpp. This version provides greater detail and clarity, especially for display modes using large fonts.

Key differences:

- Characters are stored as 13 rows of 16-bit hexadecimal values (not 15), each row encoding up to 15 pixels.
- Fonts follow a proportional design with carefully maintained width-to-height ratios.
- This layout helps maintain a visually consistent appearance across different characters.

We used an online matrix tool to fine-tune characters during development. This was especially useful when a character looked unbalanced or uneven on the actual display for the proportions huge shout-out to this Guide.

Just like the 7x7 set, this one uses a similar indexing function:

The index resolution logic is identical:

```
int getCharIndex15x15(char c) {
    if (c >= 'A' && c <= 'Z') return c - 'A';
    if (c >= 'a' && c <= 'z') return c - 'a' + 26;
    if (c >= '0' && c <= '9') return c - '0' + 52;
```

This allows easy switching between font sizes without requiring separate command formats or breaking display logic.

Offset Logic

To support flexible and visually appealing text rendering on our 60x15 LED matrix, we implemented two font systems: a compact **7x7 character set** and a more detailed **15x15 character set**, both optimized for dynamic and static display modes. Each font is stored in program memory (PROGMEM) to conserve RAM, and character spacing is handled through an offset chart for visual alignment.

Offset Chart

To further enhance visual quality, especially when combining letters of varying widths or vertical baselines, we introduced an **Offset Chart**. This chart contains alignment metadata for each character (**currently only in 15*15 font for proof of concept**):

- **Horizontal Spacing:** It is a function that takes in the width/rightmost binary(1) character to determine if additional spacing is needed between

characters to avoid visual collisions (e.g., between `'T'` and `'o'`).

- **Vertical Offset:** Especially important in the 15x15 font, this aligns characters such that descenders (`'g'`, `'y'`, `'p'`) are rendered lower, while uppercase letters stay vertically centered.

This logic is embedded in the **`needsSpacing()`**

```
//HERE IS THE NEW SPACING FUNCTION
bool Display::needsSpacing(char current, char next, bool useBigFont) {
  int currentIndex = useBigFont ? getCharIndex15x15(current) : getCharIndex(current);
  int nextIndex    = useBigFont ? getCharIndex15x15(next)    : getCharIndex(next);
```

and **`getYOffset()`**

```
//vertical offset for characters
  int Display::getCharVerticalOffset(char c, bool useBigFont) {
  if (!useBigFont) return 0;  // Skip shift for 7x7 font
  int index = getCharIndex15x15(c);
  if (index < 0 || index >= 75) return 0;
  return (int)pgm_read_byte(&(charVerticalOffset15x15[index]));
}
```

functions and is shared across both `scrollText` and `displayStaticText` implementations.

Remote Labelled Image