ROULETTE SIMULATOR

Nickaan Jahadi
Computer Engineering Department
California Polytechnic State University, San Luis Obispo

December 11, 2024

DESIGNED DEVICE BEHAVIOR

This device is a fully functional digital roulette game designed to simulate the experience of playing American Roulette. Players begin with a predefined set of chips, representing various denominations of money as depicted in the terminal interface. The game operates through a series of user-driven stages.

Once gameplay begins, players can choose to manage their chips by trading in or going straight to placing bets on various options, with the device dynamically updating the roulette table and chip balance after each action. When ready, players spin the wheel, which is realistically simulated on the terminal to rotate several times before landing on a winning spot.

After the spin, the device evaluates the player's bet against the winning spot, determining whether they won or lost. The terminal interface displays the results, updates chip balances, and allows players to continue betting or reset the game if all chips are lost.

SYSTEM SPECIFICATIONS

Electrical Characteristics		
Power Supply Source	USB Type-A to Mini-B Cable	
Regulated Power Supply Voltage	3.3 V (5 V Unregulated)	
Individual LED Resistance	220 Ω	
Individual LED Forward Voltage	~1.9 V	
Individual LED Forward Current	~6.36 mA	
Operational Characteristics		
LED Indicators	4 green - WIN 4 red - LOSS 1 blue, 1 yellow – WHEEL SPINNING	
Supported Chip Denominations	\$1 (white) \$5 (red) \$10 (blue) \$25 (green)	\$50 (orange) \$100 (black) \$500 (purple) \$1000 (yellow)
Supported Bet Types (and Payouts)	Straight (35 to 1) Split (17 to 1) Street (11 to 1) Basket (11 to 1) Corner (8 to 1) Top Line (6 to 1)	Double Street (5 to 1) Dozen (2 to 1) Column (2 to 1) Red/Black (1 to 1) Odd/Even (1 to 1) Low/High (1 to 1)
MCU Clock Frequency	80 MHz	
RNG Clock Frequency	48 MHz	
Interface Characteristics		
User Interface	VT100 – Compatible Terminal	
User Input	Text via keyboard	
Baud Rate	115200 bits/s	

Table 1: Roulette Simulator System Specifications

SYSTEM SCHEMATIC

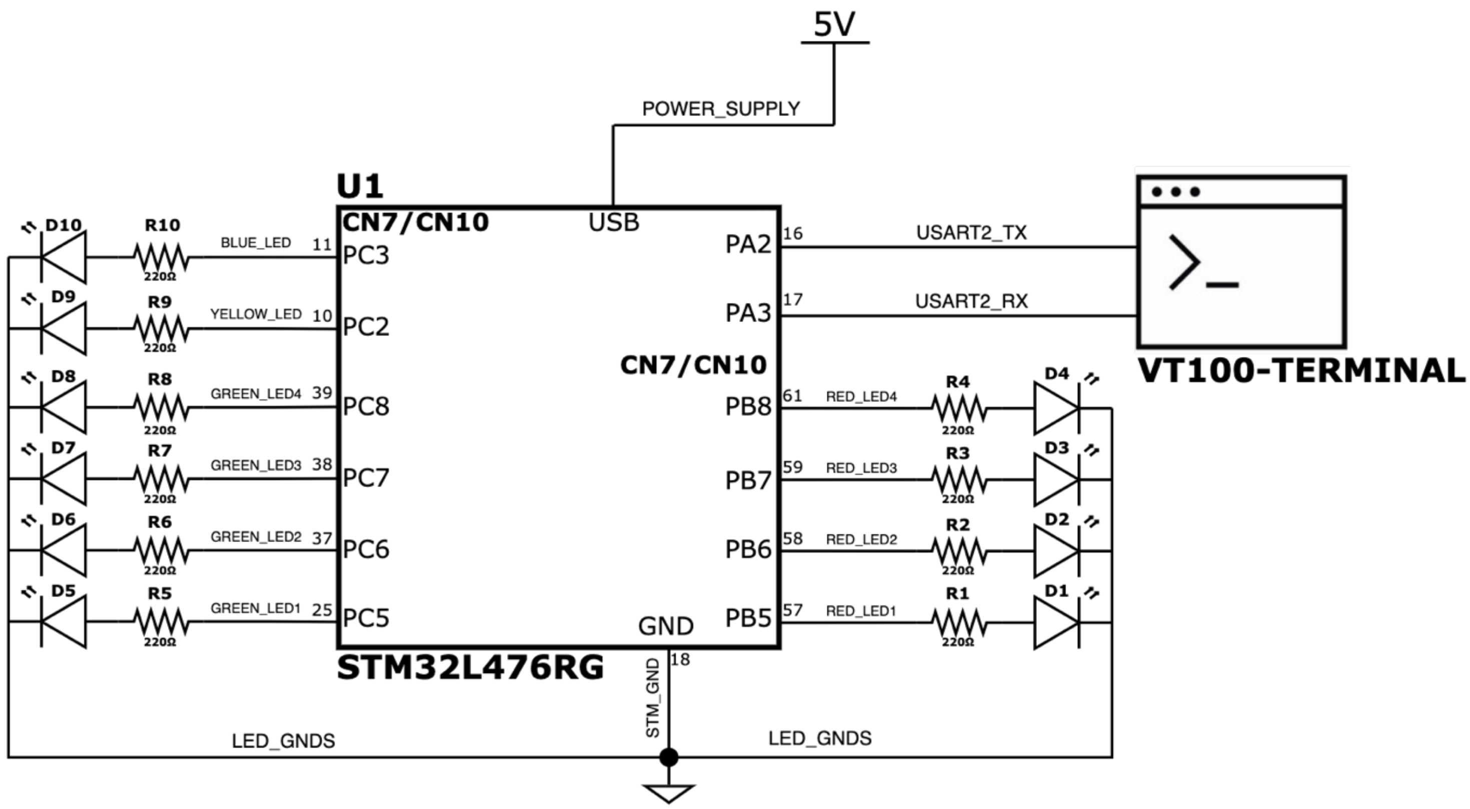

Fig. 1: Roulette Simulator Game Schematic

SOFTWARE ARCHITECTURE

OVERVIEW

The program begins by configuring the system clock to run the MCU at 80 MHz and initializing all essential peripherals required for the device's functionality. These peripherals include GPIO to control the LEDs (for visual feedback), USART2 for terminal communication, RNG for random number generation, and TIM2 for managing the timing of the spinning wheel animation.

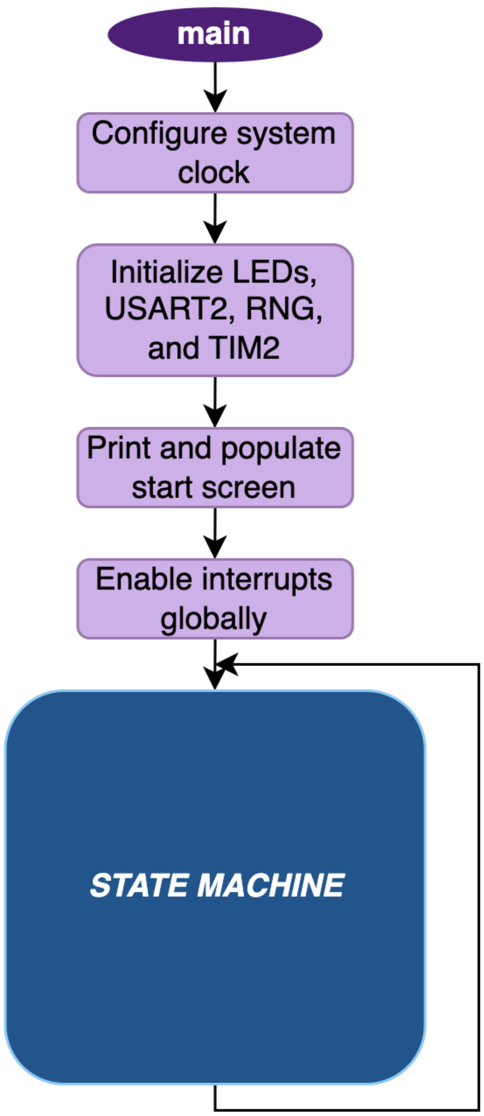

Fig. 2: main() Flowchart

Once initialization is complete, the program sets up the user interface by displaying the roulette table, spinning wheel, and the player's chip inventory on the terminal. The program utilizes VT100 escape codes extensively to allow for precise control over the terminal display, including clearing lines, positioning the cursor, and highlighting specific areas in the interface.

After setting up the interface, global interrupts are enabled to dynamically respond to events such as user inputs and timer updates. The main program then transitions into its state machine, which orchestrates the flow of the game. This state machine progresses through distinct states, such as initialization, trading chips, placing bets, spinning the wheel, displaying results, and handling end-game scenarios.

The main loop continuously repeats this cycle, managing transitions between states, monitoring new user inputs, and updating the terminal display dynamically. This structure ensures that the user can interact with the device seamlessly, adjusting their actions or inputs in real-time without disrupting the flow of gameplay.

VARIABLES

GLOBALS

This program uses several global **`#define`** constants and **`volatile`** variables in multiple functions. The constants establish fixed parameters, such as the size of lookup tables, chip denominations, and bet configurations, while the volatile variables dynamically track game states, user inputs, and chip balances.

CONSTANTS

`SINGLE_ARR_SIZE` – defines the look-up table size of single-array-based bets, which include Red, Black, Odd, Even, Low, and High. Value is 18.

`NUM_SPLITS` – defines the look-up table size of split bets. Value is 61.

`NUM_STREETS` – defines the look-up table size of street bets. Value is 12.

`NUM_BASKETS` – defines the look-up table size of basket bets. Value is 13.

`NUM_CORNERS` – defines the look-up table size of corner bets. Value is 22.

`TOP_LINE_SIZE` – defines the look-up table size of the top line bet. Value is 5.

`NUM_DUB_ST` – defines the look-up table size of double street bets. Value is 11.

`NUM_DOZ_COL` – defines the look-up table size of dozen and column bets. Value is 3.

`YELLOW_VAL` – defines the monetary value of yellow chips. Value is 1000.

`PURPLE_VAL` – defines the monetary value of purple chips. Value is 500.

`BLACK_VAL` – defines the monetary value of black chips. Value is 100.

`ORANGE_VAL` – defines the monetary value of orange chips. Value is 50.

`GREEN_VAL` – defines the monetary value of green chips. Value is 25.

`BLUE_VAL` – defines the monetary value of blue chips. Value is 10.

`RED_VAL` – defines the monetary value of red chips. Value is 5.

`WHITE_VAL` – defines the monetary value of white chips. Value is 1.

`POSSIBLE_CHIPS` – defines the total number of distinct chip denominations available in the game. Value is 8.

`ARR_SIZE` – defines the total number of spots on the roulette wheel and table. Value is 38.

`SPLIT_SIZE` – defines the number of spots included in a single split bet. Value is 2.

`ST_SIZE` – defines the number of spots included in a single street bet. Value is 3.

`BASKET_SIZE` – defines the number of spots included in a single basket bet. Value is 3.

`CORNER_SIZE` – defines the number of spots included in a single corner bet. Value is 4.

`DUB_ST_SIZE` – defines the number of spots included in a single double street bet. Value is 6.

`DOZ_COL_SIZE` – defines the number of spots included in a single dozen or column bet. Value is 12.

STRUCTURES

`typedef struct Chips` – represents the player's available chips, categorized by denomination ($1000, $500, $100, $50, $25, $10, $5, $1), and stores the quantity of each chip type.

`typedef struct Spot` – represents a single spot on the roulette table, including its associated color and number.

GAME VARIABLES

`char usart_input_buffer[20]` – stores user input received via USART in the form of a character array.

`uint8_t usart_input_index` – tracks the current position in the **`usart_input_buffer`** for adding new characters. Initialized to 0.

`bool input_ready` – a flag indicating when the user has completed input, allowing the program to process it. Initialized to false.

`typedef enum GameState` – defines the various states that the program can cycle through. Each state is assigned a unique identifier:

INIT_ST = 0, ***TRADE_ST*** = 1, ***BET_TYPE_ST*** = 2, ***TABLE_UPDATE_ST*** = 3, ***BET_MONEY_ST*** = 4, ***SPIN_ST*** = 5, ***RESULT_ST*** = 6, ***END_ST*** = 7

`GameState current_state` – tracks the current state of the game within the state machine. State machine transitions between states to manage program functionality. Initialized to ***INIT_ST***.

`uint32_t winning_index` – the index of the winning spot on the roulette wheel. Determined using the RNG. Initialized to 0.

`char winning_numbers[ARR_SIZE][3]` – an array that holds the winning numbers for the current round of bets.

`uint8_t winning_numbers_count` – tracks how many numbers are stored in the **`winning_numbers`** array. Initialized to 0.

`uint8_t spin_iterations` – counts the number of full iterations the roulette wheel makes during a spin. Initialized to 0.

`uint8_t spin_index` – tracks the current position of the roulette wheel during the spin. Initialized to 0.

`bool spin_complete` – a flag that indicates when the wheel has finished spinning. Initialized to false.

PLAYER VARIABLES

`Chips player_chips` – a structure containing the quantities of each type of chip available to the player. Initialized to:

Yellow ($1000) = 0	Green ($25) = 12
Purple ($500) = 1	Blue ($10) = 10
Black ($100) = 5	Red ($5) = 16
Orange ($50) = 10	White ($1) = 20

`uint32_t bet_amount` – tracks the total amount of chips wagered by the player in the current round. Initialized to 0.

`char bet_type[20]` – stores the type of bet the player has chosen in the form of a character array.

LOOK-UP TABLES

`Spot wheel_arr[ARR_SIZE]` – represents the layout of the roulette wheel, storing the color and number for each spot on the wheel in its unique order.

`Spot base_table_arr[ARR_SIZE]` – represents the layout of the roulette betting table, mapping each spot to its corresponding position on the table.

`char *split_bets[][SPLIT_SIZE]` – look-up table containing all possible split bets, where each entry defines two adjacent numbers on the table.

`char *street_bets[][ST_SIZE]` – look-up table containing all possible street bets, where each entry represents three consecutive numbers in a row.

char *basket_bets[][BASKET_SIZE] – look-up table containing all possible basket bets, where each entry represents three adjacent numbers (that include at least one zero).

char *corner_bets[][CORNER_SIZE] – look-up table containing all possible corner bets, where each entry represents four numbers that form a square on the table.

char *top_line[] – look-up table containing the top line bet (00, 0, 1, 2, 3).

char *double_street_bets[][DUB_ST_SIZE] – look-up table containing all possible double street bets, where each entry represents six numbers across two consecutive rows.

char *dozen_bets[][DOZ_COL_SIZE] – look-up table containing all possible dozen bets, where each entry contains twelve consecutive numbers.

char *column_bets[][DOZ_COL_SIZE] – look-up table containing all possible column bets, where each entry contains twelve numbers from one of the vertical columns on the table.

char *red[] – look-up table containing all red spots on the roulette table.

char *black[] – look-up table containing all black spots on the roulette table.

char *odds[] – look-up table containing all odd spots on the roulette table.

char *evens[] – look-up table containing all even spots on the roulette table.

char *low_half[] – look-up table containing numbers in the lower half of the roulette table (1-18).

char *high_halfs[] – look-up table containing numbers in the upper half of the roulette table (19-36).

LOCALS

misc.c

ARR_VAL – defines the auto-reload register (ARR) value for TIM2 to generate a timer update event after every full spin of the roulette wheel (1/38 s). Value is 2105263.

RNG_MULT – defines the multiplier used in the clock configuration for the RNG. Value is 24.

usart.c

BAUD_RATE – specifies the USART communication baud rate (bits/s). Value is 115200.

ESCAPE CODES – encapsulates the various ANSI escape codes used to format the VT100 terminal output for improved organization and readability.

STATE MACHINE

The state machine serves as the backbone of the program's control flow, ensuring that the game operates in a structured and sequential manner. It organizes the program into distinct states, each dedicated to a specific stage of gameplay, such as initializing the system, trading chips, placing bets, spinning the wheel, and displaying results. Transitions between states are triggered by user inputs, game events, or predefined conditions, enabling a smooth progression through the game. This structure allows the system to dynamically handle user interactions and game logic without disrupting the overall functionality.

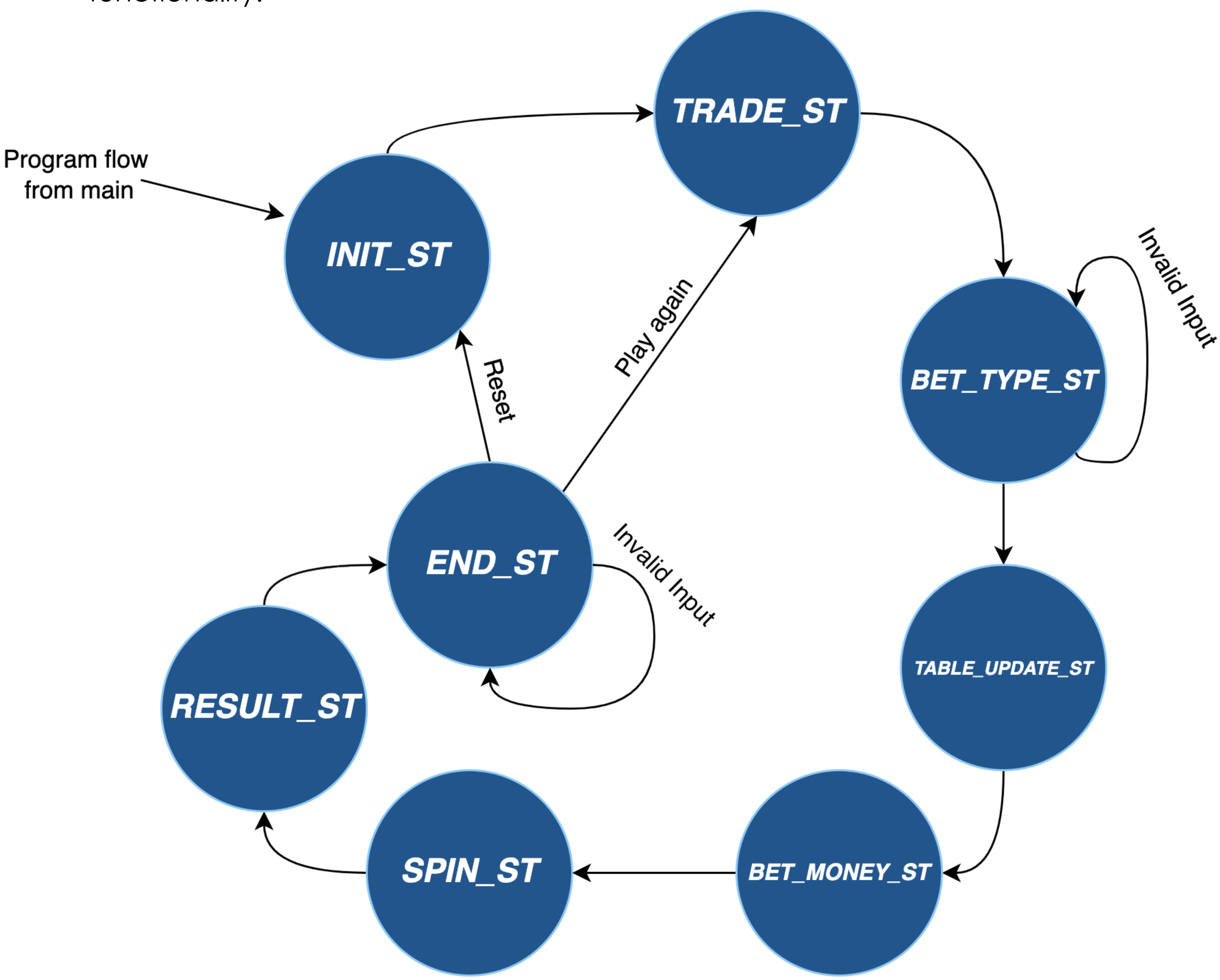

Fig. 3: State Machine State Diagram

STATES

INIT_ST

This state is the starting point of the program. It introduces the user to the game and sets the stage for future gameplay.

The program begins by displaying a welcome message on the terminal interface. The message invites the user to press the "Enter" key to initiate the game. During this phase, the program waits for the user's input, monitoring the **`input_ready`** flag, which is triggered when a complete input is received via the USART. Once the "Enter" key is detected, the program clears the welcome message and displays a brief notification indicating that the game is starting. A short delay is implemented to provide a smoother user experience, allowing the user time to process the transition message. After these actions are completed, the program transitions to ***TRADE_ST***, where the user can begin trading chips.

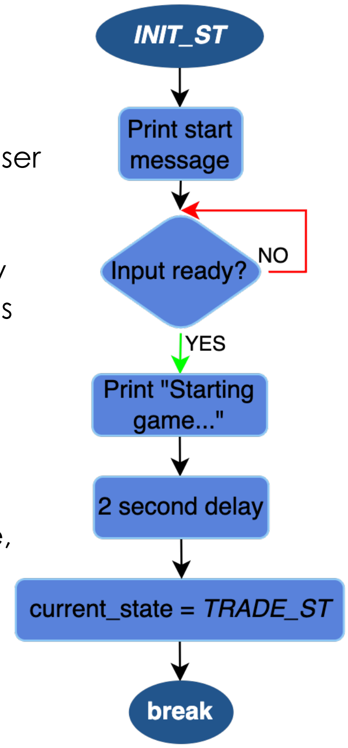

Fig. 4: *INIT_ST* Flowchart

TRADE_ST

This state allows the user to exchange their higher-value chips for lower-value chips. It begins by asking the user whether they want to trade in chips, presenting a simple "yes" or "no" prompt via the terminal. If the user enters "no," the state immediately transitions to ***BET_TYPE***, skipping the chip trade-in process entirely.

If the user chooses "yes," the program prompts them to specify the chip value they wish to trade in. Several checks are conducted at this stage to ensure the validity of the input: the chip value must be a valid denomination, the user must have enough chips of the specified denomination, and certain trades (e.g., trading in $1 chips) are prohibited. Once the program validates the chip type, it asks the user how many of those chips they wish to trade in, verifying that the quantity does not exceed their available chips.

Next, the user is prompted for the chip value they want in return. The system checks that the new chip value is lower than the traded-in chip value and that the trade value is divisible without

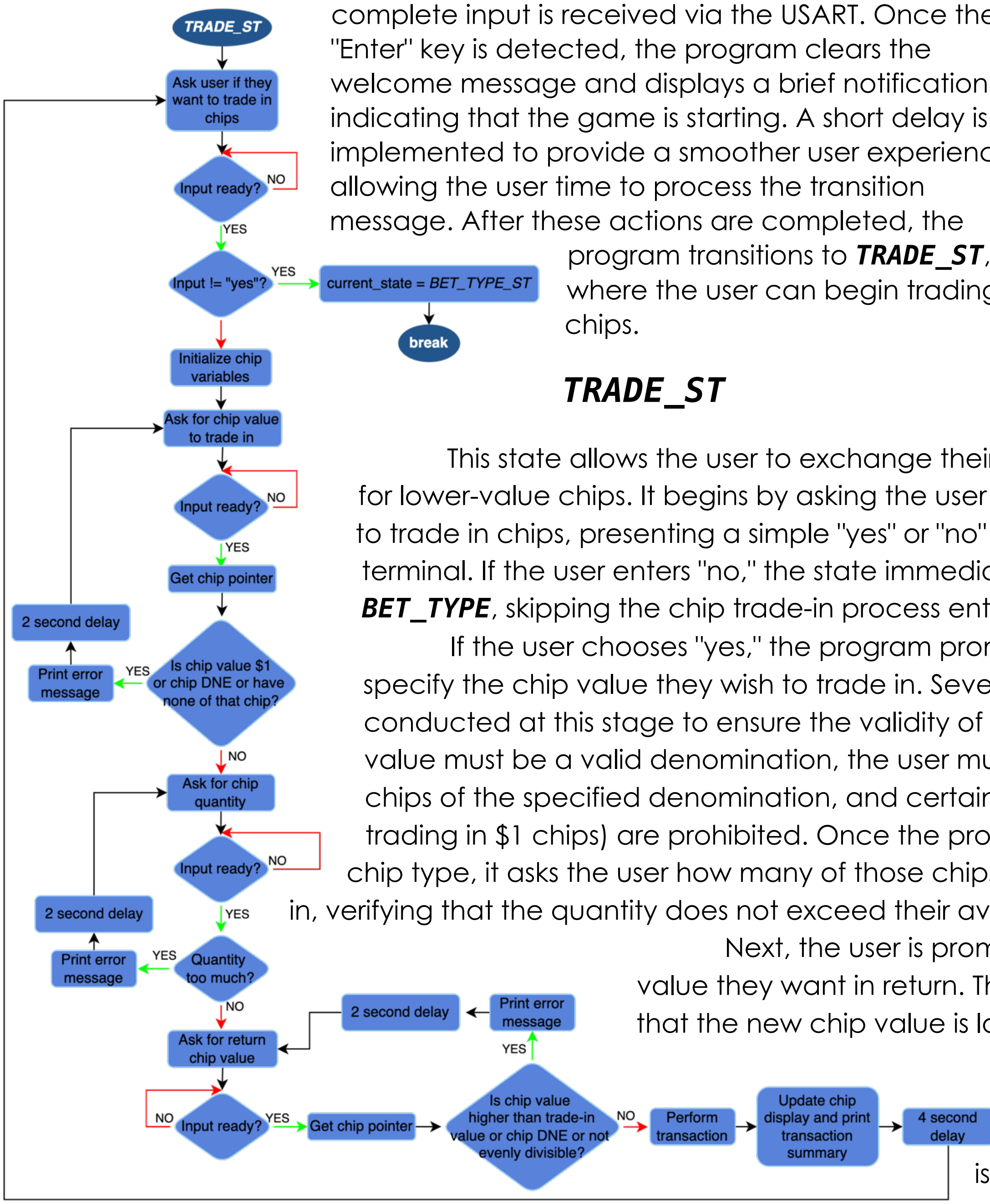

Fig. 5: *TRADE_ST* Flowchart

remainder, ensuring a fair exchange. Once all inputs are validated, the number of new chips the user will receive is calculated, their chip inventory is updated, and the results of the trade are displayed in a clear message.

The user can repeat this process as many times as they like, with the state returning to the initial prompt after each successful trade. The state ensures that invalid inputs are flagged with appropriate error messages, allowing the user to correct their choices. Once the user finishes trading, they can type "no" to transition into ***BET_TYPE_ST***, where they can start placing bets.

BET_TYPE_ST

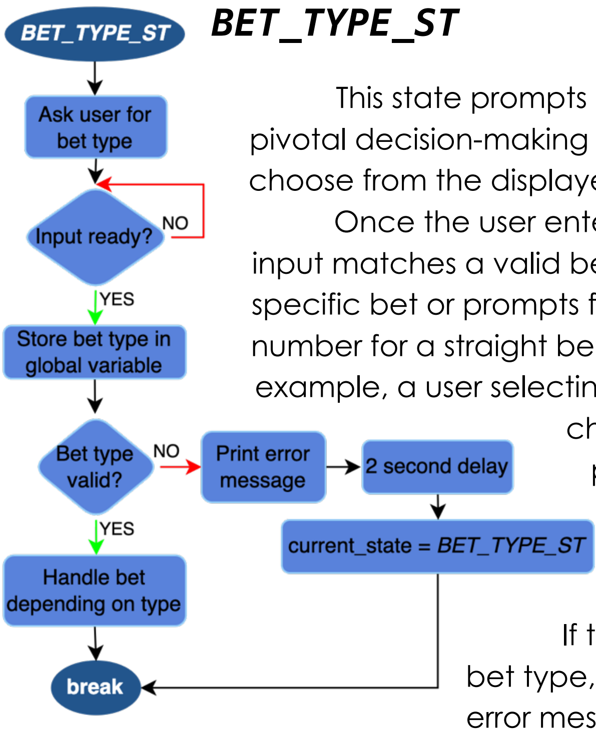

Fig. 6: *BET_TYPE_ST* Flowchart

This state prompts the user to select a type of bet to place, serving as a pivotal decision-making point in the roulette process. It enables users to choose from the displayed list of valid bet types.

Once the user enters their choice, the program validates the input. If the input matches a valid bet type, the system either transitions to handle the specific bet or prompts for additional information, such as selecting a specific number for a straight bet or a row index for bets like splits or streets. For example, a user selecting a "Split" bet will be directed to another function to choose the desired row from a predefined lookup table. Similarly, "Red" or "Black" bets automatically highlight the corresponding slots. If the user enters an invalid bet type, the program displays an error message and remains in the state to allow the user to re-enter their choice. Once the bet is successfully validated and processed, the program transitions to ***TABLE_UPDATE_ST***, where it updates the display to reflect the selected bet.

TABLE_UPDATE_ST

This state updates the roulette betting table to visually reflect the selected bets. It dynamically generates a new version of the table by copying the predefined base table and applying visual highlights to the spots associated with the user's bets, making them clearly distinguishable.

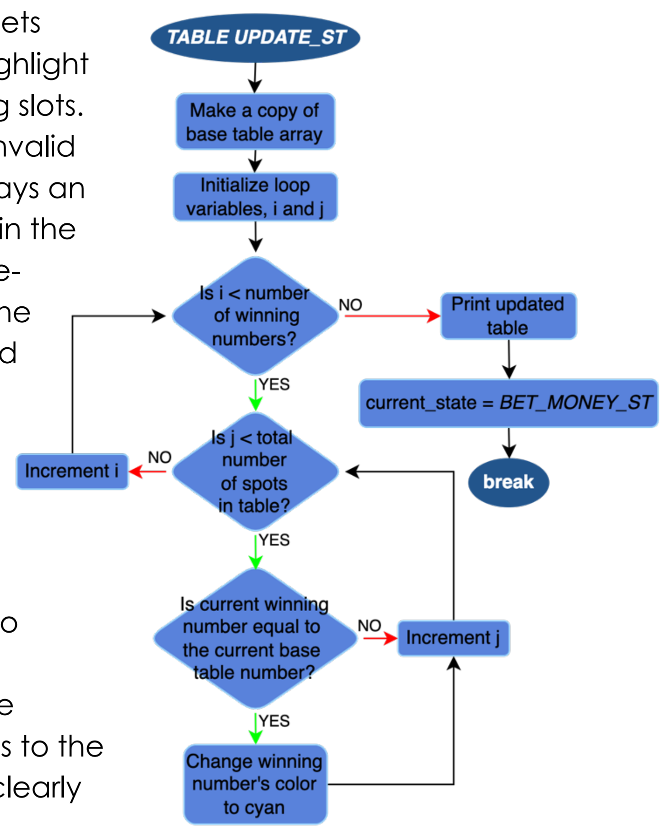

Fig. 7: *TABLE_UPDATE_ST* Flowchart

The program iterates through the user's selected winning numbers and compares each number with the numbers in the table. Using string comparison, it identifies the matching spots and changes their color attribute to cyan. This serves as a visual cue, helping users confirm the areas of the table they have bet on.

After highlighting the appropriate spots, the updated dynamic table is displayed on the terminal using the **`USART_print_table`** function. Once the table has been updated and shown to the user, the program transitions to ***`BET_MONEY_ST`***, where users can allocate chips to their selected bets.

`BET_MONEY_ST`

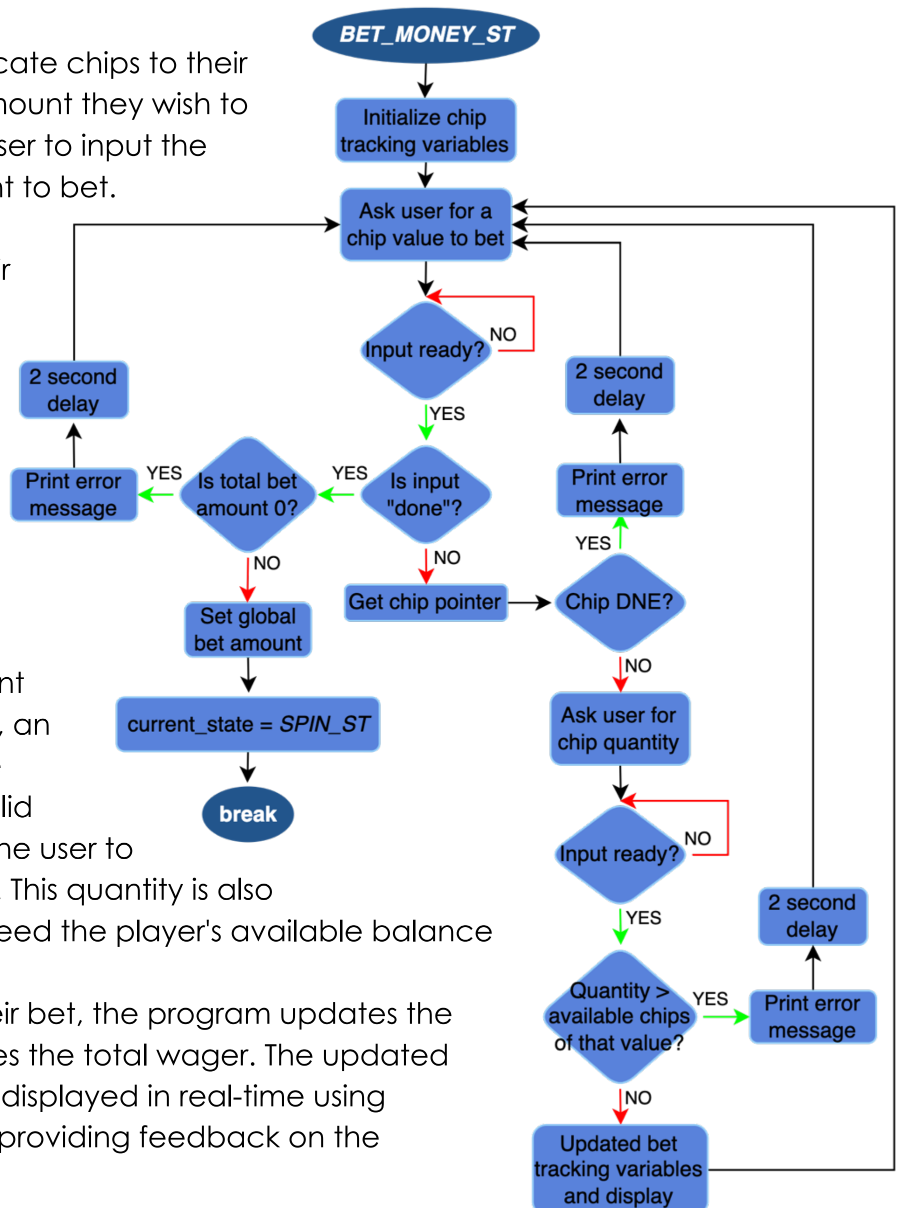

Fig. 8: *BET_MONEY_ST* Flowchart

This state allows users to allocate chips to their selected bets, defining the total amount they wish to wager. The program prompts the user to input the denomination of the chip they want to bet. The user can either enter a chip value, or type "done" to finalize their bet. If "done" is entered without betting any chips, the program issues an error message instructing the user to place a bet before proceeding.

Once a valid chip value is provided, the program validates it against the player's available chips. If the input chip value is invalid, or if the player lacks sufficient chips of the entered denomination, an error message is displayed, and the system requests a new input. For valid inputs, the program then prompts the user to specify the quantity of chips to bet. This quantity is also validated to ensure it does not exceed the player's available balance of that chip denomination.

As the user adds chips to their bet, the program updates the player's chip count and recalculates the total wager. The updated chip balance and bet amount are displayed in real-time using the **`USART_print_chips`** function, providing feedback on the player's remaining resources.

The state operates in a continuous loop, allowing users to add multiple chip denominations to their bet. When the user is satisfied with their wager and types "done," the total bet amount is stored in the global variable **`bet_amount`**. The program then transitions to ***SPIN_ST***, where the roulette wheel is spun to determine the outcome of the bet.

SPIN_ST

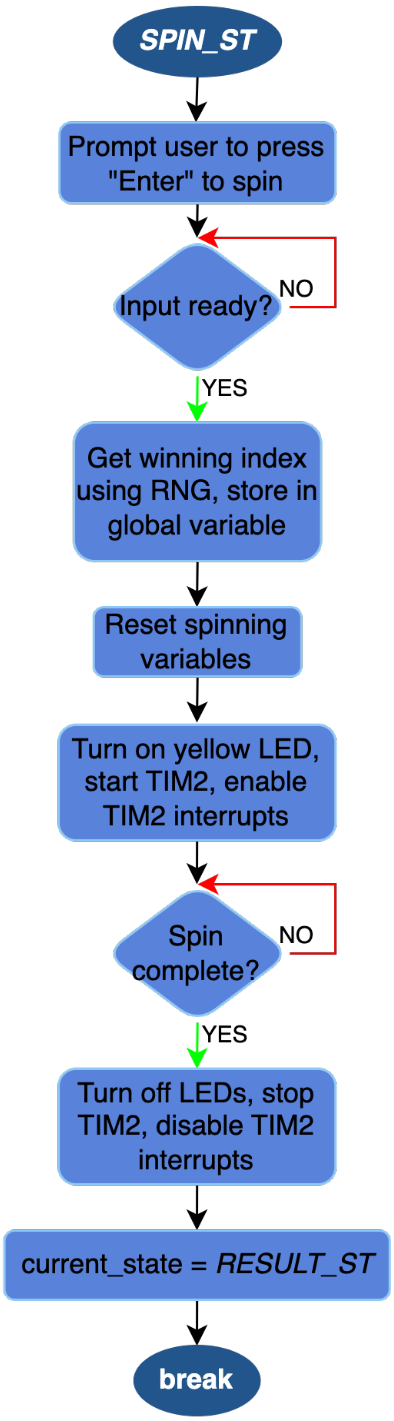

Fig. 9: *SPIN_ST* Flowchart

This state is where the wheel is spun to determine the winning spot. The program prompts the user to press the "Enter" key to initiate the spin. Once the user provides the input, the system utilizes the random number generator (RNG) peripheral to generate a random index corresponding to one of the 38 spots on the roulette wheel (**`winning_index`**).

The spinning animation is achieved through a combination of hardware-based timers and LEDs. The program begins by enabling the TIM2 timer, which triggers periodic updates to the terminal display to simulate the wheel spinning. Simultaneously, the yellow and blue LEDs alternately flash, creating a visual effect that mimics the dynamic movement of a physical roulette wheel.

During the spinning process, **`spin_index`** is incremented cyclically, iterating through the wheel's array until **`winning_index`** is reached. The spinning continues for several iterations before gradually stopping at the randomly determined winning spot. This delay is controlled using the **`spin_iterations`** variable to make the process more realistic. Once the wheel lands on the winning spot, the **`spin_complete`** flag is set to true, and the LEDs are turned off to signify the end of the spin.

After the spin concludes, the program transitions to ***RESULT_ST*** to evaluate the spin's outcome, compare it with the user's bets, and display the results.

RESULT_ST

This state determines the spin's outcome and calculates the user's winnings or losses based on their bets. The program first resets the roulette table to its default, unhighlighted state. Next, it retrieves the winning spot from the **`wheel_arr`** array based on the previously determined **`winning_index`**. To determine if the user has won, the program iterates through the **`winning_numbers`** array, which holds the user's selected spots. It compares each entry with the winning spot number, and if a match is found, the user has won.

The program then calculates the net gain or loss for the user. If the user has won, the **`calculate_odds`** function determines the payout multiplier for the bet type, and the winnings are distributed back to the user's chip balance using the **`distribute_chips`** function. Conversely, if the user loses, the **`bet_amount`** is deducted as their net loss.

After determining the outcome, a result message is prepared and displayed in the terminal. If the user has won, a congratulatory message is shown, and the green LEDs are illuminated to enhance the celebratory effect. If the user loses, a message encouraging them to try again is displayed, with the red LEDs indicating the loss. The resulting message includes details about the amount won or lost.

Finally, the bet_amount is reset to zero, and the chip balance display is updated in the terminal. The program then transitions to ***END_ST***, where the user can decide to continue playing or end the game.

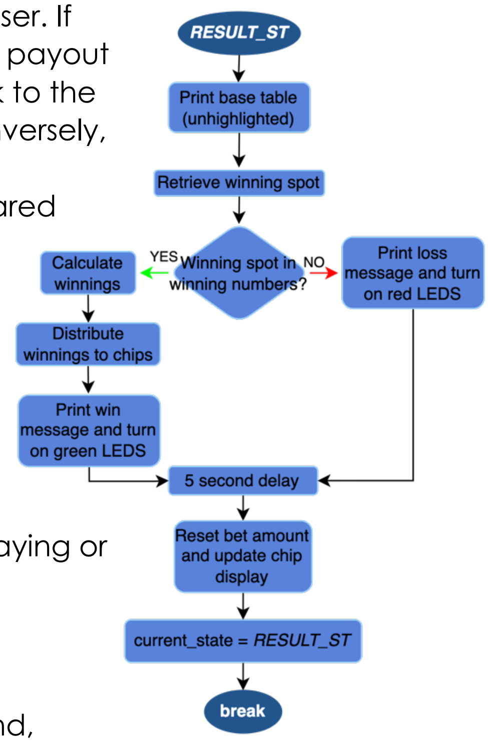

Fig. 10: *RESULT_ST* Flowchart

END_ST

This state is the conclusion of each game round, ensuring that all game-related data is reset and preparing the program for the next round or restarting the game. The program first clears all bet-related variables. It resets the **`bet_type`** string, clears the **`winning_numbers`** array, and resets **`winning_numbers_count`**.

Next, the program evaluates the player's chip balance using the **`calculate_total_balance`** function. If the user's total balance is zero, the terminal displays a message informing them that they are out of chips and prompts them to type "reset" to restart the game. The program then waits for user input. If the user types "reset," the **`player_chips`** structure is reinitialized to its default values, representing the starting chip distribution. The chip display is updated in the terminal using **`USART_print_chips`**, and the program transitions back to ***INIT_ST***, effectively restarting the game.

If the user's input is invalid or does not match "reset," the program remains in ***END_ST***, prompting the user again for valid input.

If the user has chips remaining, the terminal displays a message inviting the user to press "Enter" to play another round. The program waits for the user to press Enter and transitions to ***TRADE_ST***, allowing them to trade chips before placing new bets.

Finally, all LEDs are turned off to signify the end of the round, providing visual feedback that the game state has reset.

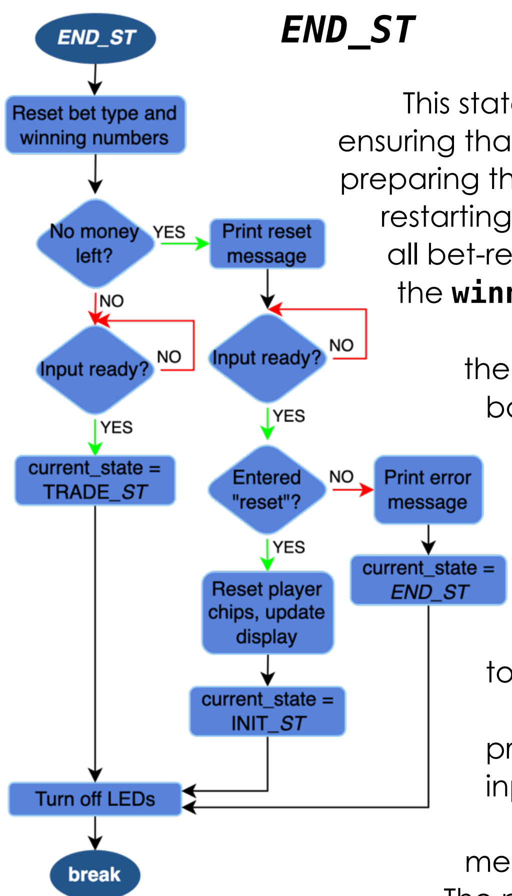

Fig. 11: *END_ST* Flowchart

FUNCTIONS

TIMER FUNCTIONS

TIM2_init

This function initializes the TIM2 timer to control the timing of the wheel spinning. After enabling the TIM2 clock, it configures the timer to count upward and sets the auto-reload register to **ARR_VAL** – 1, which corresponds to the desired timing for one full wheel spin over one second. Interrupts are then enabled for the auto-reload event, which triggers when the timer reaches its set period. Any existing interrupt flags are cleared before starting the timer to avoid unintentional behavior. Finally, it enables the timer with the Nested Vectored Interrupt Controller (NVIC) [1].

TIM2_IRQHandler

This function handles the interrupt service routine (ISR) for TIM2, which governs the spinning of the roulette wheel during ***SPIN_ST***. This ISR is triggered when the timer reaches its update event.

First, it clears the update flag to prevent repeated triggers from the same event. If the spin is still in progress, it increments the **spin_index**, cycling through the wheel positions stored in **wheel_arr**. This simulates the visual spinning of the roulette wheel by calling **USART_print_wheel** to update the terminal display dynamically. The function also toggles the yellow and blue LEDs at a visible rate to enhance the spinning effect.

To determine the end of the spin, the function checks if the wheel has completed at least five full rotations and the **spin_index** matches the **winning_index**. If these conditions are met, the **spin_complete** flag is set to true, signaling the spin's completion. Each full wheel rotation is tracked by incrementing **spin_iterations** when **spin_index** gets back to 0.

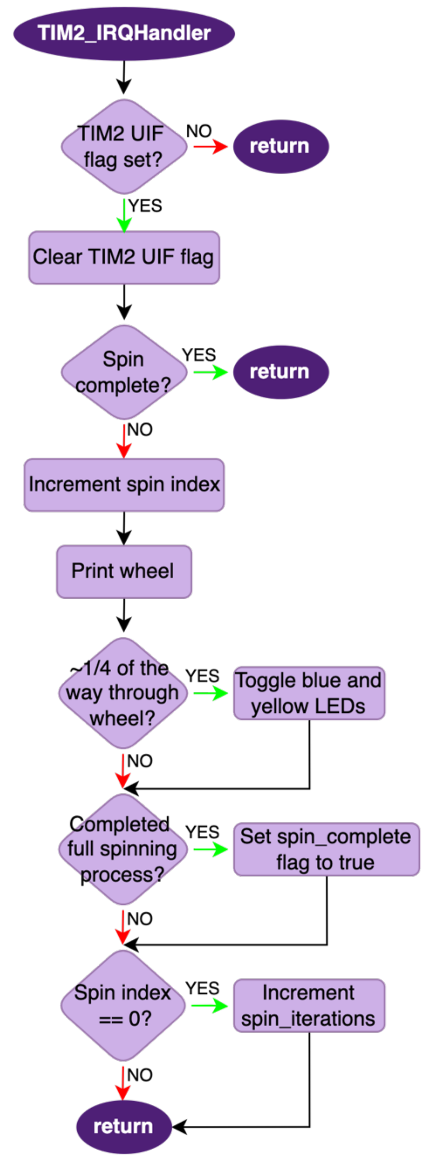

Fig. 12: TIM2_IRQHandler() Flowchart

USART FUNCTIONS

USART_init

This function configures and enables the USART peripheral for serial communication. It begins by enabling the clock for GPIO port A and configuring pins PA2 and PA3 for alternate functions (USART transmit and receive, respectively) [2]. These pins are set to high-speed mode with no pull-up or pull-down resistors. The function assigns the alternate function for USART2 (function 7) to these pins and enables the clock for the USART2 peripheral. Next, the USART word length, stop bits, and parity are configured for standard 8-bit data transmission with 1 stop bit and no parity. The USART divisor in the Baud Rate Register (BRR) is set by dividing the system clock frequency (80 MHz) by the desired baud rate (115200 bits/s). To enable USART functionality, the transmit and receive enable bits are set, and receiver interrupts are activated to allow for asynchronous communication. Finally, the USART peripheral is enabled, completing its initialization [1].

USART2_IRQHandler

This function is the ISR for the USART2 peripheral. It handles incoming serial data from the terminal, processes user inputs, and manages the input buffer. When a character is received, the function first checks the RXNE flag to ensure that data is ready to be read. The character is then fetched from the Receive Data Register and categorized based on its type.

If the character is a backspace, the function processes it by removing the last character from the input buffer and updating the cursor on the terminal display. The cursor moves back, prints a space to erase the character visually, and then moves back again to maintain alignment.

If the character represents the "Enter" key, the function considers the input complete. It null-terminates the buffer to form a valid string, resets the buffer index to zero, and sets the **`input_ready`** flag to true, signaling that the input is available for processing by the main program.

For other characters, the function appends the character to the input buffer, provided there is sufficient space remaining. The character is also echoed back to the terminal so the user can visualize their input.

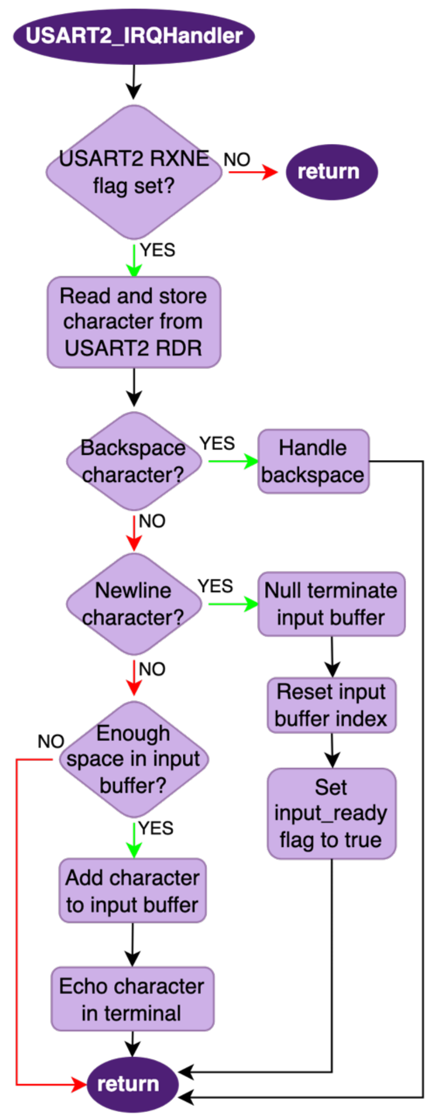

Fig. 13: USART2_IRQHandler() Flowchart

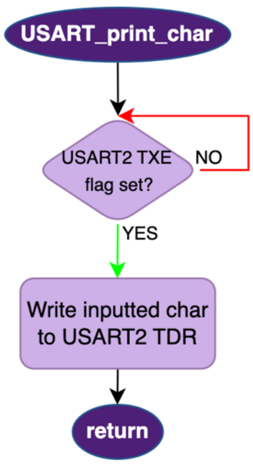

Fig. 14: USART_print_char() Flowchart

`USART_print_char`

This function facilitates the transmission of a single character to the terminal via USART. It waits until the USART transmit data register (TDR) is empty, indicated by the transmit data register empty (TXE) flag, and then loads the character into the TDR. This ensures the character is transmitted without overwriting previous data.

`USART_print_string`

This function enables the transmission of a null-terminated string to the terminal via USART. It iterates through the string, character by character, and calls **`USART_print_char`** to send each character. This process continues until the null character (\0) is encountered, signaling the end of the string.

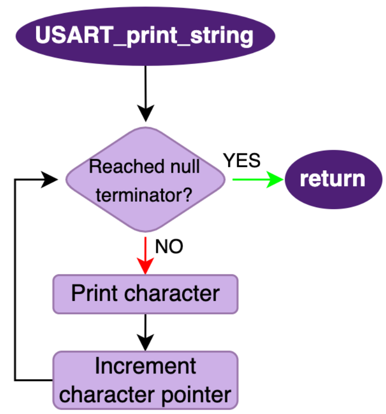

Fig. 15: USART_print_string() Flowchart

`USART_ESC_Code`

This function is used to send ANSI escape codes to the terminal. Escape codes allow for controlling terminal behavior, such as clearing the screen, hiding the cursor, or moving the cursor to specific positions, enabling dynamic updates to the display.

It takes a string representing an escape code (e.g., "[2J" for clearing the screen) as its argument. It first sends the ESC character ("\x1B") to the terminal, indicating the beginning of an escape sequence. Then, it transmits the provided escape code string using **`USART_print_string`**.

`USART_reset_screen`

This function resets the terminal display and prepares it for a clean and organized output by sending a series of ANSI escape codes using **`USART_ESC_Code`**. The escape codes clear the screen, reposition the cursor to the top-left corner of the terminal, reset the text's attributes, and hide the cursor.

`USART_print_line`

This function prints a single line of text, such as a segment of the roulette table or wheel outline, to the terminal interface. It ensures that the printed line is followed by a cursor movement to the next line for proper formatting. First, the function outputs the provided string line to the terminal using the **`USART_print_string`** function. It then sends an escape code to move the cursor one line down (DOWN_1) and another escape code to position the cursor to the far left (FULLY_LEFT). These escape codes ensure that the printed content is correctly aligned and that subsequent outputs start from the beginning of the next line.

USART_print_section

This function efficiently displays multi-line content on the terminal interface by iterating through an array of strings and printing each line in sequence. It takes two parameters: a pointer to an array of strings where each string represents a line to be printed, and an integer that specifies the number of lines in the array. Within the function, a loop iterates through each string in the array, passing it to the **USART_print_line** function. This subfunction is responsible for printing the string and managing terminal cursor movement to ensure proper alignment and spacing.

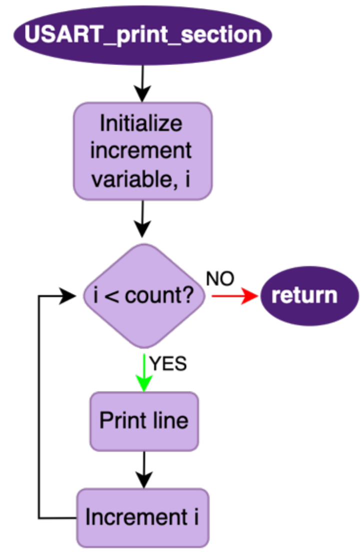

Fig. 16: USART_print_section() Flowchart

USART_start_screen

This function initializes and displays the game's startup screen. This screen includes the game title, the roulette wheel outline, the betting table, a bottom container section with information on payouts and chips, and a dynamic display of chip denominations in their respective colors.

It begins by resetting the terminal using **USART_reset_screen**. It then prints the title at a centered position using bold and underlined text, followed by resetting text attributes to default. Next, it utilizes the **USART_print_section** function to display the predefined outlines for the roulette wheel, table, and a bottom container that organizes the layout into distinct sections.

To enhance user understanding, the chip values are displayed in their respective colors. Using escape codes, the function positions the cursor precisely and applies color attributes to match each chip's denomination. The alignment and spacing between chip values ensure the visual clarity of the layout.

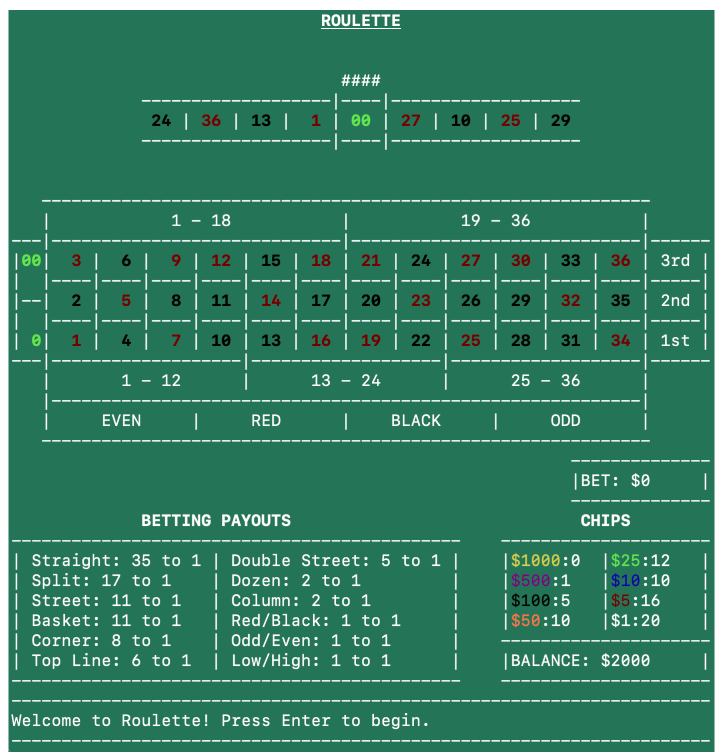

Fig. 17: Roulette Starting Screen

USART_print_table

This function displays the roulette betting table numbers on the terminal interface, with the spots formatted in their correct positions and appropriately color-coded. It starts by navigating the terminal cursor to the correct position, setting the output location to align with the table's starting position. The "00" spot, which is the topmost position on the table, is printed first. If the spot's color is marked as "cyan", indicating it is a winning spot, the text is displayed in cyan. Otherwise, it is displayed in green, its default color.

Next, the function iterates through the table's numbered spots, which are indexed from 2 to 37 (skip “00” and “0”). For each spot, the function determines its color based on its color attribute. Red spots are displayed in red, black spots in black, and highlighted winning spots in cyan. The numbers are spaced evenly to ensure proper alignment, maintaining a clean and visually intuitive layout.

At the end of each row (specifically after the 13th and 25th spots), the function adjusts the cursor to the next row's starting position by moving left, down, and right by predefined steps. This ensures the table's grid structure is preserved. Finally, the "0" spot is printed below the "00" spot, with similar color-coding logic applied. Once all spots are printed, the function resets the terminal's text attributes to default, ensuring no residual styling affects subsequent output.

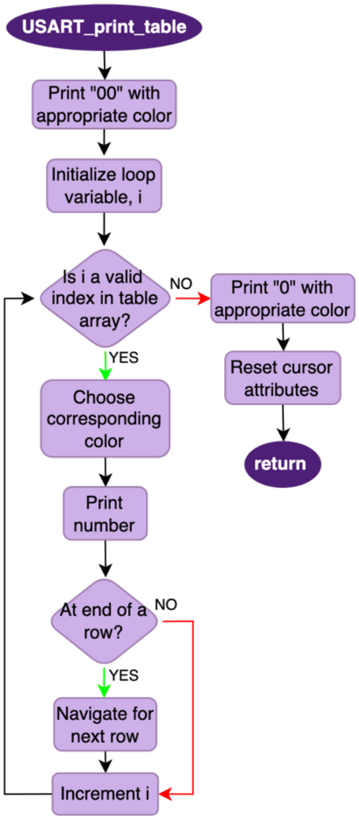

Fig. 18: USART_print_table() Flowchart

`USART_print_wheel`

This function displays the roulette wheel numbers on the terminal interface, with the spots formatted in their correct positions and appropriately color-coded. It starts by calculating the size of the wheel “window” (9) and determining the appropriate starting index. It then navigates the terminal cursor to align with the wheel’s starting position. Utilizing the calculated starting index and window size, the winning spot is determined and stored.

Next, the function iterates through nine wheel spots to populate the window. For each spot, the loop determines its index in the **`wheel_arr`**, sets it color based on the spot’s color attribute, and prints the number. The numbers are spaced evenly to ensure proper alignment. After printing the entire wheel window, the function returns the winning spot.

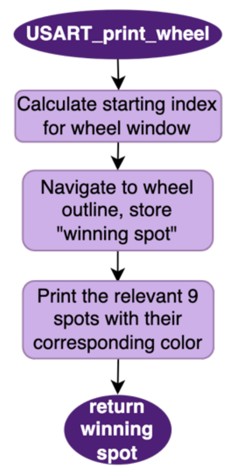

Fig. 19: USART_print_wheel() Flowchart

`USART_print_balance`

This function displays the player’s current balance on the terminal. It starts by initializing a character buffer to store the string representation of the balance. After navigating to the specific position in the chip display for the balance, the previous balance is cleared by iteratively overwriting the characters with spaces, ensuring no interference with residual values. The inputted balance is then converted to a string and printed to the display.

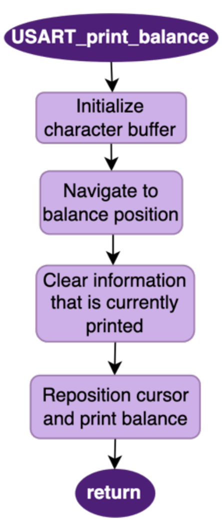

Fig. 20: USART_print_balance() Flowchart

`USART_print_bet`

This function displays the player's current bet on the terminal. Similar to **`USART_print_balance`**, it starts by initializing a character buffer to store the string representation of the bet. After navigating to the specific position in the chip display for the bet, the previous bet is cleared by overwriting the characters with spaces. The inputted bet is then converted to a string and printed to the display.

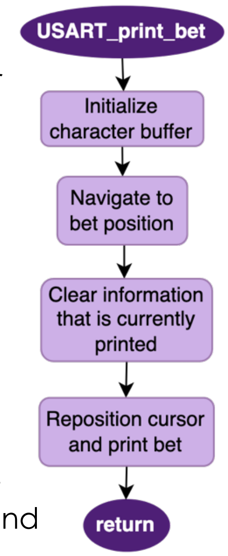

Fig. 21: USART_print_bet() Flowchart

`USART_print_chips`

This function updates the display of the player's chip counts, balance, and bet amount in the terminal interface. It begins by declaring a character array to store a string representation of the chip count. Then, it defines and utilizes the **`USART_clear_and_print`** helper function to navigate the terminal and print the updated chips counts for all the corresponding chip values. Once all chip values are updated, the total balance is calculated based on the new number of chips, and the total bet and balance are printed to the display.

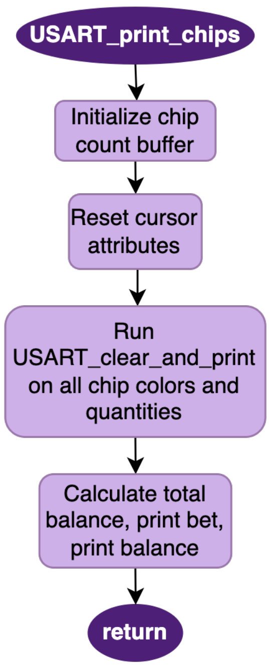

Fig. 22: USART_print_chips() Flowchart

`USART_clear_and_print`

This is a helper function defined within **`USART_print_chips`** to efficiently update specific areas of the terminal display. It takes in three parameters: down, right, and value. The "down" parameter represents the number of lines to move the cursor down from the top left corner, while the "right" parameter represents the number of characters to move the cursor right from the current position. "Value" is the integer value to be printed at the desired location. Once the cursor is positioned in the proper place, the function overwrites the previous characters at that position and moves the cursor back to the beginning of the cleared area. The integer value is then converted into a string and stored in the **`USART_print_chips`** character array, eventually being printed to the terminal.

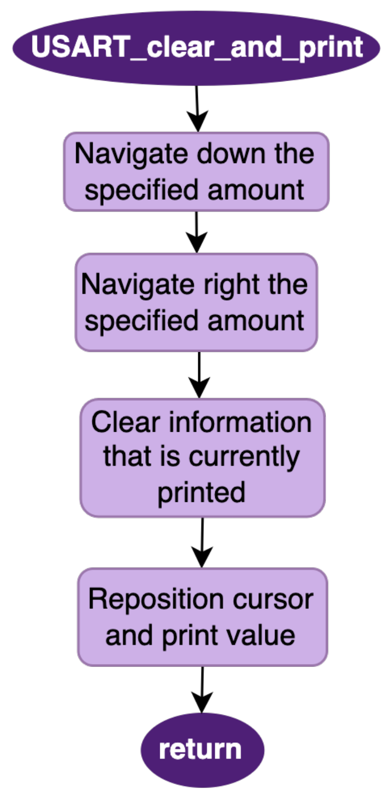

Fig. 23: USART_clear_and_print() Flowchart

MISCELLANEOUS FUNCTIONS

`LED_init`

This function configures the LEDs that serve as visual indicators during the program's operation, providing feedback during spinning and result announcements. It begins by enabling the GPIOB and GPIOC clocks. The function then configures pins 5 – 8 on GPIOB and GPIOC and pins 2 and 3 on only GPIOC as outputs, ensuring they operate in push-pull mode with very high-speed settings and no pull-up or pull-down resistors. All the LED pins are initialized to a low state, ensuring that all LEDs are off when the program starts [1].

`RNG_init`

This function initializes the Random Number Generator (RNG) peripheral to produce random numbers, which are critical for simulating the roulette wheel's random spin outcomes. It configures the RNG to operate with a 48 MHz clock derived from the PLLSAI1Q output. It starts by disabling the PLLSAI1 to allow reconfiguration. It then sets the multiplier for PLLSAI1 to the predefined value **`RNG_MULT`** (24), ensuring the PLL generates a clock frequency suitable for the RNG. After the PLLSAI1Q output is enabled and the PLLSAI1 is restarted, the clock source for the RNG is configured to use the PLLSAI1Q output. Finally, the RNG clock is enabled and the RNG itself is activated [1].

`RNG_get_random_number`

This function generates a random number using the hardware random number generator (RNG) peripheral. It first checks the status register for any error flags, such as seed error or clock error. If any of these flags are set, it clears them to reset the error state, ensuring the RNG can produce valid results. Next, the function waits for the data ready flag to be set. This flag indicates that a new random number is available in the RNG's data register. Finally, the function retrieves and returns the random number stored in the RNG's data register, providing a 32-bit hardware-generated random value.

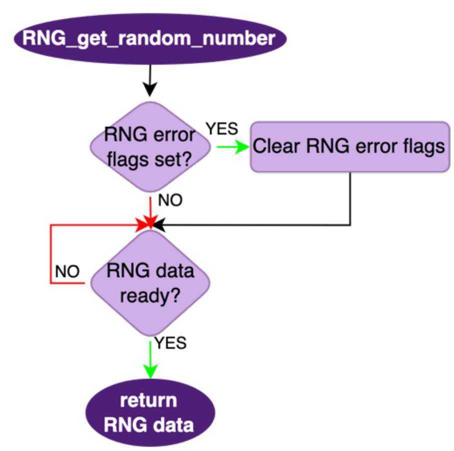

Fig. 24: RNG_get_random_number() Flowchart

`calculate_total_balance`

This function computes the total monetary value of a player's chips based on their quantities and respective denominations. It takes a **`Chips`** structure as input, which contains the counts of each type of chip the player possesses. It multiplies the number of chips of each denomination by their corresponding value, defined as constants. The function then sums these individual contributions to calculate the player's total balance. Finally, the function returns the total balance, representing the player's overall chip value.

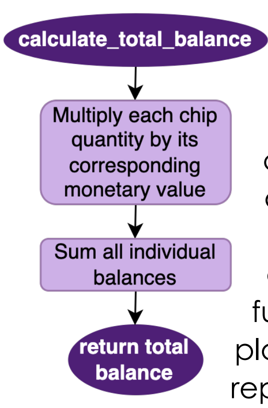

Fig. 25: calculate_total_balance() Flowchart

`calculate_odds`

This function determines the payout multiplier associated with a specific bet type. It accepts a string pointer that specifies the type of bet the player has placed. It compares this input against predefined bet types, and depending on the match, it returns the corresponding payout multiplier. If the bet type does not match any known category, the function returns 0, indicating invalid or unrecognized input.

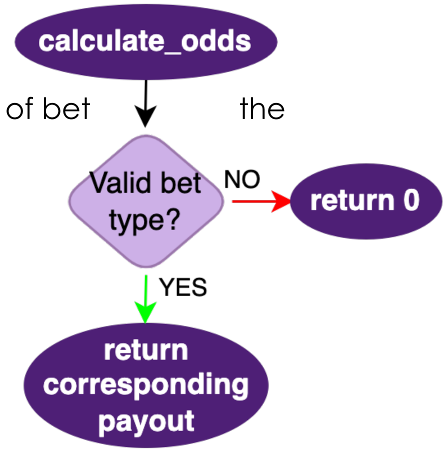

Fig. 26: calculate_odds() Flowchart

`distribute_chips`

This function converts the given monetary amount into chips of various denominations and updates the player's chip counts accordingly. It begins by defining two arrays to track the monetary values of each chip denomination and the pointers to the respective chip count variables within the **`Chips`** structure, allowing the function to directly update these values.

Using a loop, the function iterates through each chip denomination, starting with the highest. For each denomination, it repeatedly checks if the remaining amount is greater than or equal to the current chip's value. If so, it increments the corresponding chip count, reduces the amount by the chip's value, and continues until the remaining amount is less than the current denomination. This process ensures that the distribution always prioritizes higher-value chips, minimizing the total number of chips distributed.

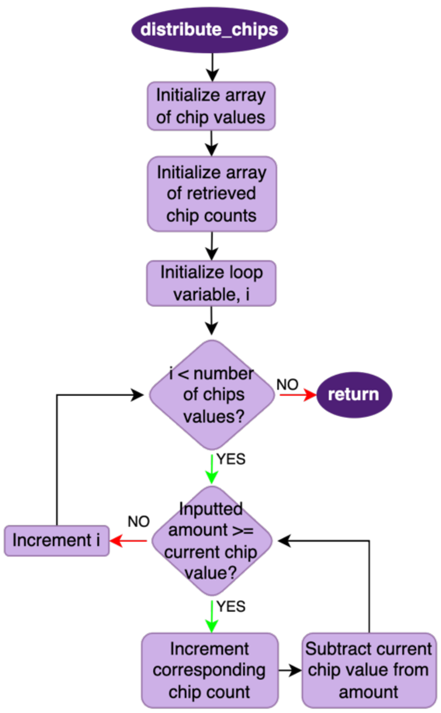

Fig. 27: distribute_chips() Flowchart

`get_chip_pointer`

This function validates a given chip value and returns a pointer to the corresponding chip count within the **`Chips`** structure. As parameters, it takes the chip value, representing the monetary value of the chip to validate, and a pointer to a **`Chips`** structure, which stores the counts for each chip denomination. Using a switch statement, the function compares the input chip value against predefined constants for each valid chip denomination.

If the chip value matches one of the predefined denominations, the function returns a pointer to the corresponding field in the **`Chips`** structure. If the input chip value does not match any of the predefined denominations, the function returns NULL, signaling that the value is invalid.

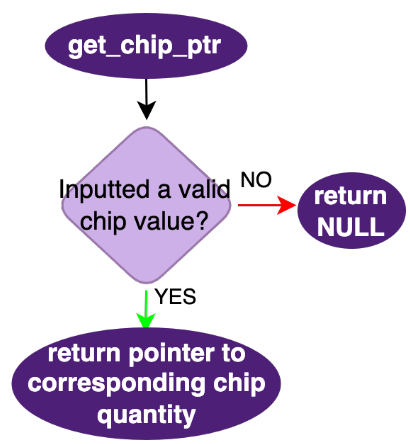

Fig. 28: get_chip_pointer() Flowchart

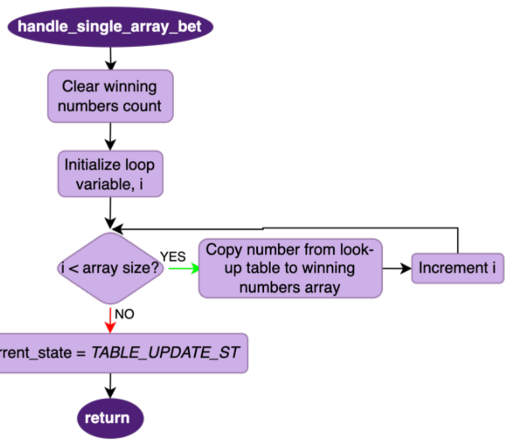

Fig. 29: handle_single_array_bet() Flowchart

handle_single_array_bet

This function processes bets that are represented by a single array of numbers. It begins by resetting the **winning_numbers_count** variable to zero, ensuring that any previous bet data is cleared. It then iterates over the provided bet array, which contains the numbers associated with the user's bet, and copies each number into the global **winning_numbers** array. The **winning_numbers_count** variable is incremented with each addition to track the total number of winning numbers for the current bet. After populating the **winning_numbers** array, the function sets the **current_state** to ***TABLE_UPDATE_ST***, signaling the state machine to proceed to the table update state.

handle_double_array_bet

This function manages bets that are represented by a two-dimensional array, where each row in the array corresponds to a distinct betting option. It begins by prompting the user to input the index of the desired bet option. It dynamically constructs a prompt using the bet name parameter to ensure clarity for the user. This prompt is displayed on the terminal, and the program waits for the user's input. Once the input is received, it is converted from a 1-based index (as the user would input) to a 0-based index for internal processing.

The function then validates the input to ensure that the selected index is within the bounds of the array. If the index is valid, the function clears the global **winning_numbers_count** and retrieves the corresponding row from the bet array using pointer arithmetic. It iterates through the row, copying each number into the **winning_numbers** array while updating the **winning_numbers_count** variable to track the total numbers added.

Upon successfully populating the **winning_numbers** array, the function transitions the **current_state** to ***TABLE_UPDATE_ST***, allowing the table to be updated to reflect the selected bet. If the input index is invalid, an error message is displayed, and the state remains in ***BET_TYPE_ST*** to let the user retry.

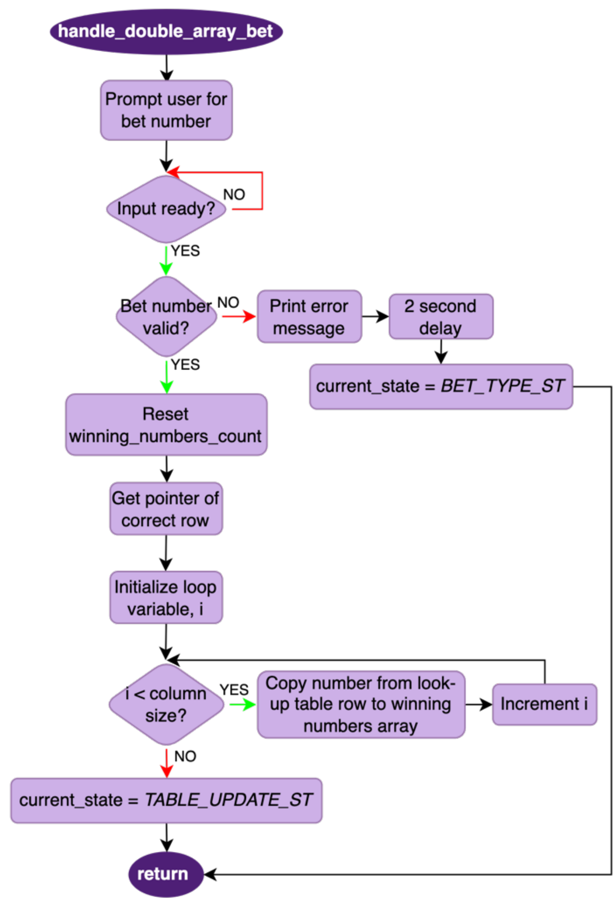

Fig. 30: handle_double_array_bet() Flowchart

APPENDICES

SOURCE CODE

main.c

```c
#include "main.h"
#include "usart.h"
#include <stdbool.h>
#include <stdlib.h>

void SystemClock_Config(void);
void handle_single_array_bet(const char **, uint8_t);
void handle_double_array_bet(char *, const char **, uint8_t, uint8_t);

#define SINGLE_ARR_SIZE 18
#define NUM_SPLITS 61 //number of possible split bets
#define NUM_STREETS 12 //number of possible street bets
#define NUM_BASKETS 3 //number of possible basket bets
#define NUM_CORNERS 22 //number of possible corner bets
#define TOP_LINE_SIZE 5
#define NUM_DUB_ST 11 //number of possible double street bets
#define NUM_DOZ_COL 3 //number of possible dozen/column bets
#define DEL 2000 //1 second in milliseconds

//game states
typedef enum {
    INIT_ST,
     TRADE_ST,
    BET_TYPE_ST,
     TABLE_UPDATE_ST,
     BET_MONEY_ST,
    SPIN_ST,
    RESULT_ST,
    END_ST
} GameState;

volatile GameState current_state = INIT_ST; //current state of the game
volatile char usart_input_buffer[20]; //buffer for USART input
volatile uint8_t usart_input_index = 0; //index for USART buffer
volatile bool input_ready = false; //flag to indicate input is complete

//player data
Chips player_chips = {
    .yellow = 0, //$1000 chips
    .purple = 1, //$500 chips
    .black = 5, //$100 chips
```

```
    .orange = 10, //$50 chips
    .green = 12, //$25 chips
    .blue = 10, //$10 chips
    .red = 16, //$5 chips
    .white = 20 //$1 chips
};

volatile uint32_t bet_amount = 0; //current bet amount
volatile char bet_type[20]; //type of bet

//game data
volatile uint32_t winning_index = 0; //winning number index
volatile char winning_numbers[ARR_SIZE][3]; //buffer to store winning spots
volatile uint8_t winning_numbers_count = 0; //count of winning numbers in buffer
volatile uint8_t spin_iterations = 0; //number of completed wheel iterations
volatile uint8_t spin_index = 0; //current wheel index during spinning
volatile bool spin_complete = false; //flag to indicate the spin is done

int main(void) {
     HAL_Init();
     SystemClock_Config();
     //initialize LEDs, USART, RNG, TIM2, and print/populate start screen
     LED_init();
     USART_init();
     RNG_init();
     TIM2_init();
     USART_start_screen();
     USART_print_wheel(wheel_arr, 0);
     USART_print_table(base_table_arr);
     USART_print_chips(&player_chips, bet_amount);
     __enable_irq(); //enable interrupts globally

     while (1) { //infinte program flow
          switch (current_state) { //state machine
               case INIT_ST: //start point of game
                    //print start message
                    USART_ESC_Code(TOP_LEFT);
                    USART_ESC_Code(DOWN_35);
                    USART_ESC_Code(CLEAR_LINE);
                    USART_print_string("Welcome to Roulette!
                                        Press Enter to begin.");

                    while (!input_ready); //wait for user input
                    input_ready = false; //reset input flag

                    USART_ESC_Code(CLEAR_LINE);
                    USART_ESC_Code(FULLY_LEFT);
                    USART_print_string("Starting game...");
                    HAL_Delay(DEL); //2 second delay
                    current_state = TRADE_ST; //transition to trading state
                    break;
```

```
case TRADE_ST: //handle chip trade in logic
     while(1) {
          //ask user if they want to trade in chips
          USART_ESC_Code(CLEAR_LINE);
          USART_ESC_Code(FULLY_LEFT);
          USART_print_string("Trade in chips? (yes/no) --> ");

          while (!input_ready); //wait for user input
          input_ready = false; //reset input flag

          //transition to betting type state if the
          //answer is not "yes"
          if (strcmp(usart_input_buffer, "yes") != 0) {
               //transition to betting type state
               current_state = BET_TYPE_ST;
               break;
          }
        //ask the user for the type of chip to trade in
        uint32_t chip_value_in = 0;
        uint32_t chip_value_out = 0;
        uint32_t chip_quantity_in = 0;
        uint32_t possible_out = 0;
        uint32_t *chip_ptr_in = NULL;
        uint32_t *chip_ptr_out = NULL;

        //prompt for the chip to trade in
        while (1) {
            USART_ESC_Code(CLEAR_LINE);
            USART_ESC_Code(FULLY_LEFT);
            USART_print_string("Enter chip value to
                                trade in --> ");

            while (!input_ready); //wait for user input
            input_ready = false; //reset input flag

            //validate and record chip value
            chip_value_in = atoi(usart_input_buffer);
            //disallow trading in white chips
            if (chip_value_in == WHITE_VAL) {
               USART_ESC_Code(CLEAR_LINE);
               USART_ESC_Code(FULLY_LEFT);
               USART_print_string("Cannot trade in $1 chips!
                                   Please enter a higher chip
                                   value.");
               HAL_Delay(DEL); //2 second delay
               continue; //retry for valid input
            }
            //assign pointer to corresponding chip value
            //in player chips
            chip_ptr_in =
            get_chip_pointer(chip_value_in, &player_chips);
            //handle an invalid input
```

```
        if (chip_ptr_in == NULL) {
          USART_ESC_Code(CLEAR_LINE);
          USART_ESC_Code(FULLY_LEFT);
          USART_print_string("Invalid chip value! Please
                              enter a valid chip value.");
          HAL_Delay(DEL); //2 second delay
          continue; //retry for valid input
        } else if (*chip_ptr_in == 0) { //check if out of
                                         //those chips
          USART_ESC_Code(CLEAR_LINE);
          USART_ESC_Code(FULLY_LEFT);
          USART_print_string("You are out of ");
          snprintf(usart_input_buffer,
          sizeof(usart_input_buffer), "$%lu",
          chip_value_in);
          USART_print_string(usart_input_buffer);
          USART_print_string(" chips! Please enter a
                              different chip value.");
          HAL_Delay(DEL); //2 second delay
          continue; //retry for valid input
        }
        break;
    }

      //ask the user how many of these chips to trade in
      while (1) {
            USART_ESC_Code(CLEAR_LINE);
            USART_ESC_Code(FULLY_LEFT);
            USART_print_string("Enter quantity of ");
            snprintf(usart_input_buffer,
            sizeof(usart_input_buffer), "$%lu",
            chip_value_in);
            USART_print_string(usart_input_buffer);
            USART_print_string(" chips to trade in --> ");

            while (!input_ready); //wait for user input
            input_ready = false; //reset input flag

            //validate chip quantity
            chip_quantity_in = atoi(usart_input_buffer);
            if (chip_quantity_in > *chip_ptr_in) {
                  USART_ESC_Code(CLEAR_LINE);
                  USART_ESC_Code(FULLY_LEFT);
                  USART_print_string("Not enough chips to
                                      trade in! Please enter a
                                      lower quantity.");
                  HAL_Delay(DEL); //2 second delay
                  continue;
            }
            break;
      }
```

```
//ask the user for the chip value they want in return
while (1) {
    USART_ESC_Code(CLEAR_LINE);
    USART_ESC_Code(FULLY_LEFT);
    USART_print_string("Enter chip value you want
                       in return --> ");

    while (!input_ready); //wait for user input
    input_ready = false; //reset input flag

    //validate and record chip value
    chip_value_out = atoi(usart_input_buffer);
    //validate that the chip value out is lower
    //than chip value in
    if (chip_value_out >= chip_value_in) {
        USART_ESC_Code(CLEAR_LINE);
        USART_ESC_Code(FULLY_LEFT);
        USART_print_string("Chip value must be lower
                           than trade-in chip value!
                           Try again.");
        HAL_Delay(DEL); //2 second delay
        continue;
    }
    //assign pointer to corresponding chip value
    //being traded in for
    chip_ptr_out =
    get_chip_pointer(chip_value_out, &player_chips);
    //handle invalid input
    if (chip_ptr_out == NULL) {
       USART_ESC_Code(CLEAR_LINE);
       USART_ESC_Code(FULLY_LEFT);
       USART_print_string("Invalid chip value! Please
                          enter a valid chip value.");
       HAL_Delay(DEL); //2 second delay
       continue; //retry for valid input
    }
    //validate that the lower value fits evenly
    //into the higher value
    if (((chip_value_in * chip_quantity_in) %
          chip_value_out) != 0) {
        USART_ESC_Code(CLEAR_LINE);
        USART_ESC_Code(FULLY_LEFT);
        USART_print_string("Trade-in value must be
                           divisible by the desired
                           value! Try again.");
        HAL_Delay(DEL); //2 second delay
        continue;
    }
    break;
}
```

```
            //calculate the possible number of lower chips
            possible_out =
            (chip_quantity_in * chip_value_in) / chip_value_out;
            //perform the transaction
            *chip_ptr_in -= chip_quantity_in;
            *chip_ptr_out += possible_out;

            //update the display
            USART_print_chips(&player_chips, bet_amount);
            //notify the user of the transaction
            USART_ESC_Code(TOP_LEFT);
            USART_ESC_Code(DOWN_35);
            USART_ESC_Code(CLEAR_LINE);
            USART_ESC_Code(FULLY_LEFT);
            USART_print_string("Trade in complete! You traded ");
            snprintf(usart_input_buffer, sizeof(usart_input_buffer),
                    "%lu", chip_quantity_in);
            USART_print_string(usart_input_buffer);
            USART_print_string(" $");
            snprintf(usart_input_buffer, sizeof(usart_input_buffer),
                    "%lu", chip_value_in);
            USART_print_string(usart_input_buffer);
            USART_print_string(" chips for ");
            snprintf(usart_input_buffer, sizeof(usart_input_buffer),
                    "%lu", possible_out);
            USART_print_string(usart_input_buffer);
            USART_print_string(" $");
            snprintf(usart_input_buffer, sizeof(usart_input_buffer),
                    "%lu", chip_value_out);
            USART_print_string(usart_input_buffer);
            USART_print_string(" chips.");
            HAL_Delay(2 * DEL); //4 second delay
        }

    case BET_TYPE_ST: //determine the type of bet the user wants
        //ask for the type of bet
        USART_ESC_Code(CLEAR_LINE);
        USART_ESC_Code(FULLY_LEFT);
        USART_print_string("Choose your bet type
                            (from above) --> ");

        while (!input_ready); //wait for user input
        input_ready = false; //reset input flag

        //store chosen bet type in a global variable for later use
        strncpy(bet_type, usart_input_buffer,
        sizeof(bet_type) - 1);
        //ensure null termination
        bet_type[sizeof(bet_type) - 1] = '\0';
        if (strcmp(bet_type, "Straight") == 0) { //straight bet
            //ask for the number
            USART_ESC_Code(CLEAR_LINE);
```

```
        USART_ESC_Code(FULLY_LEFT);
        USART_print_string("Enter number (00-36) --> ");

        while (!input_ready); //wait for user input
        input_ready = false; //reset input flag

        //validate number and update the winning numbers array
        if (strcmp(usart_input_buffer, "00") == 0 ||
            (atoi(usart_input_buffer) >= 0 &&
            atoi(usart_input_buffer) <= 36)) {
                winning_numbers_count = 0;
                strncpy(winning_numbers[winning_numbers_count++],
                usart_input_buffer, 3);
                //move to betting amount state
                current_state = TABLE_UPDATE_ST;
        } else {
                //invalid number
                USART_ESC_Code(CLEAR_LINE);
                USART_ESC_Code(FULLY_LEFT);
                USART_print_string("Invalid bet! Number you
                                    entered does not exist on
                                    the wheel.");
                HAL_Delay(DEL); //2 second delay
                //remain in betting type state
                current_state = BET_TYPE_ST;
        }
} else if (strcmp(bet_type, "Split") == 0) { //split bet
        handle_double_array_bet("split", split_bets,
        NUM_SPLITS, SPLIT_SIZE);
} else if (strcmp(bet_type, "Street") == 0) { //street bet
        handle_double_array_bet("street", street_bets,
        NUM_STREETS, ST_SIZE);
} else if (strcmp(bet_type, "Basket") == 0) { //basket bet
        handle_double_array_bet("basket", basket_bets,
        NUM_BASKETS, BASKET_SIZE);
} else if (strcmp(bet_type, "Corner") == 0) { //corner bet
        handle_double_array_bet("corner", corner_bets,
        NUM_CORNERS, CORNER_SIZE);
} else if (strcmp(bet_type, "Top Line") == 0) { //top line
        handle_single_array_bet(top_line, TOP_LINE_SIZE);
} else if (strcmp(bet_type, "Double Street") == 0) {
//double street bet
        handle_double_array_bet("double street",
        double_street_bets, NUM_DUB_ST, DUB_ST_SIZE);
} else if (strcmp(bet_type, "Dozen") == 0) { //dozen bet
        handle_double_array_bet("dozen", dozen_bets,
        NUM_DOZ_COL, DOZ_COL_SIZE);
} else if (strcmp(bet_type, "Column") == 0) { //column bet
        handle_double_array_bet("column", column_bets,
        NUM_DOZ_COL, DOZ_COL_SIZE);
} else if (strcmp(bet_type, "Red") == 0 || strcmp(bet_type,
        "Black") == 0) { //color bet
```

```
            const char **selected_color =
            (strcmp(bet_type, "Red") == 0)
                                          ? red
                                          : black;
            handle_single_array_bet(selected_color,
            SINGLE_ARR_SIZE);
        } else if (strcmp(bet_type, "Odd") == 0 || strcmp(bet_type,
                  "Even") == 0) { //odd/even bet
          const char **selected_parity =
          (strcmp(bet_type, "Odd") == 0)
                                        ? odds
                                        : evens;
          handle_single_array_bet(selected_parity,
          SINGLE_ARR_SIZE);
        } else if (strcmp(bet_type, "Low") == 0 || strcmp(bet_type,
                   "High") == 0) { //high/low bet
          const char **selected_range =
          (strcmp(bet_type, "Low") == 0)
                                         ? low_half
                                         : high_half;
          handle_single_array_bet(selected_range, SINGLE_ARR_SIZE);
        } else {
            //invalid bet type
            USART_ESC_Code(CLEAR_LINE);
            USART_ESC_Code(FULLY_LEFT);
            USART_print_string("Invalid bet type! Choose one from
                                the list above.");
            HAL_Delay(DEL); //2 second delay
            //remain in betting type state
            current_state = BET_TYPE_ST;
        }
        break;

    case TABLE_UPDATE_ST:
        //create a dynamic array to copy the base table spots
        Spot dynamic_table[ARR_SIZE];
        memcpy(dynamic_table, base_table_arr,
        sizeof(base_table_arr));

        //highlight the winning spots in the dynamic array
        for (uint8_t i = 0; i < winning_numbers_count; i++) {
          for (uint8_t j = 0; j < ARR_SIZE; j++) {
                if (strcmp(winning_numbers[i],
                    dynamic_table[j].number +
                     (dynamic_table[j].number[0] == ' ' ? 1 : 0))
                      == 0) {
                       //highlight winning spot
                       strcpy(dynamic_table[j].color, "cyan");
                }
          }
        }
```

```
        //print the updated table
        USART_print_table(dynamic_table);
        //transition to betting money state
        current_state = BET_MONEY_ST;
        break;

    case BET_MONEY_ST:
        //variables to track bet info
        uint32_t chip_value = 0;
        uint32_t chip_quantity = 0;
        uint32_t total_bet = 0;

        //run continuously while in this state
        while (1) {
            //ask for chip value or "done"
            USART_ESC_Code(TOP_LEFT);
            USART_ESC_Code(DOWN_35);
            USART_ESC_Code(CLEAR_LINE);
            USART_print_string("Enter chip value to bet or
                               'done' --> ");

            while (!input_ready); //wait for user input
            input_ready = false; //reset input flag

            //check if the user is done betting
            if (strcmp(usart_input_buffer, "done") == 0) {
                if (total_bet == 0) {
                    USART_ESC_Code(CLEAR_LINE);
                    USART_ESC_Code(FULLY_LEFT);
                    USART_print_string("You must bet before
                    spinning the wheel!");
                    HAL_Delay(DEL); //2 second delay
                    continue; //retry for valid input
                }
                break;
            }

            //validate and record chip value
            chip_value = atoi(usart_input_buffer);
            uint32_t *chip_ptr = NULL;

            chip_ptr =
            get_chip_pointer(chip_value, &player_chips);

         if (chip_ptr == NULL) {
           USART_ESC_Code(CLEAR_LINE);
           USART_ESC_Code(FULLY_LEFT);
           USART_print_string("Invalid chip value! Please enter a
                              valid chip value.");
           HAL_Delay(DEL); //2 second delay
           continue; //retry for valid input
         }
```

```
            //ask for the quantity of the selected chip
            USART_ESC_Code(CLEAR_LINE);
            USART_ESC_Code(FULLY_LEFT);
            USART_print_string("Enter quantity of chips --> ");

            while (!input_ready); //wait for user input
            input_ready = false; //reset input flag

            //validate chip quantity
            chip_quantity = atoi(usart_input_buffer);
            if (chip_quantity > *chip_ptr) {
                  //invalid chip quantity
                  USART_ESC_Code(CLEAR_LINE);
                  USART_ESC_Code(FULLY_LEFT);
                  USART_print_string("Not enough chips! Please
                                      enter a lower quantity or
                                      different value.");
                  HAL_Delay(DEL); //2 second delay
                  continue;
            }
            //calculate the total bet and update chip counts
            total_bet += chip_value * chip_quantity;
            *chip_ptr -= chip_quantity;
            //update chip and balance display
            USART_print_chips(&player_chips, total_bet);
      }
      //update global bet amount
      bet_amount = total_bet;

      current_state = SPIN_ST; //transition to spin state
      break;

case SPIN_ST:
      //prompt user to press enter to spin the wheel
      USART_ESC_Code(CLEAR_LINE);
      USART_ESC_Code(FULLY_LEFT);
      USART_print_string("Press Enter to spin the wheel...");

      while (!input_ready); //wait for user input
      input_ready = false; //reset input flag

      //message while spinning
      USART_ESC_Code(CLEAR_LINE);
      USART_ESC_Code(FULLY_LEFT);
      USART_print_string("Spinning...");
      //get winning index using RNG
      winning_index = RNG_get_random_number() % ARR_SIZE;
      //reset spinning variables
      spin_iterations = 0;
      spin_index = 0;
      spin_complete = false;
```

```
        //turn on yellow LED to alternate with blue
        GPIOC->ODR |= YELLOW_PIN;
        //enable the timer to start spinning
        TIM2->CR1 |= TIM_CR1_CEN; //start timer
        TIM2->DIER |= TIM_DIER_UIE; //enable update interrupt
        TIM2->SR &= ~TIM_SR_UIF; //clear update flag

        //wait for spin to complete
        while (!spin_complete);

        GPIOC->ODR &= ~(YELLOW_PIN | BLUE_PIN); //turn off LEDs
        //disable the timer to stop spinning
        TIM2->CR1 &= ~TIM_CR1_CEN; //stop timer
        TIM2->DIER &= ~TIM_DIER_UIE; //disable update interrupt
        TIM2->SR &= ~TIM_SR_UIF; //clear update flag

        current_state = RESULT_ST; //transition to result state
        break;

    case RESULT_ST:
        //reset the table to unhighlighted values
        Spot unhighlighted_table[ARR_SIZE];
        memcpy(unhighlighted_table, base_table_arr,
               sizeof(base_table_arr));
        USART_print_table(unhighlighted_table);
        //retrieve the winning spot
        Spot winning_spot = wheel_arr[winning_index];
        //check if the winning spot is in the winning numbers array
        bool user_won = false;
        for (uint8_t i = 0; i < winning_numbers_count; i++) {
          if (strcmp(winning_numbers[i], winning_spot.number +
              (winning_spot.number[0] == ' ' ? 1 : 0)) == 0) {
                user_won = true;
                break;
          }
        }
        //calculate winnings or losses
        uint32_t net_gain = 0; //net gain or loss
        if (user_won) {
             //calculate winnings based on bet type odds
             uint32_t winnings =
             bet_amount * calculate_odds(bet_type);
             net_gain = winnings; //net gain
             //distribute winnings back as chips
             distribute_chips(winnings, &player_chips);
        } else {
             net_gain = bet_amount; //net loss
        }
        //prepare result message
        char result_message[25];
        if (user_won) {
          snprintf(result_message, sizeof(result_message),
```

```
         "You won $%ld! ", (net_gain - bet_amount));
        } else {
          snprintf(result_message, sizeof(result_message),
          "You lost $%ld. ", net_gain);
        }
        //reset bet amount
        bet_amount = 0;
        //update the chips and balance display
        USART_print_chips(&player_chips, bet_amount);
        //navigate to message section
        USART_ESC_Code(TOP_LEFT);
        USART_ESC_Code(DOWN_35);
        USART_ESC_Code(CLEAR_LINE);
        USART_ESC_Code(FULLY_LEFT);
        USART_ESC_Code(RESET_ATTRIBUTES);
        //inform the user of the result
        if (user_won) {
             USART_print_string("Congratulations! ");
             GPIOC->ODR |= LED_PINS;
        } else {
             USART_print_string("Better luck next time! ");
             GPIOB->ODR |= LED_PINS;
        }
        USART_print_string(result_message);
        HAL_Delay(2.5 * DEL); //5 second delay

        current_state = END_ST; //transition to end state

  case END_ST:
      //reset bet-related variables
      //clear the bet type
      memset(bet_type, 0, sizeof(bet_type));
      //clear the winning numbers array
      memset(winning_numbers, 0, sizeof(winning_numbers));
      winning_numbers_count = 0; //reset winning numbers count
      //check if the player is out of chips
      if (calculate_total_balance(player_chips) == 0) {
          //inform the user that they are out of chips
          USART_ESC_Code(TOP_LEFT);
          USART_ESC_Code(DOWN_35);
          USART_ESC_Code(CLEAR_LINE);
          USART_ESC_Code(FULLY_LEFT);
          USART_print_string("You are out of chips! Type 'reset' to
                               start over --> ");

             while (!input_ready); //wait for user input
             input_ready = false; //reset input flag

             //if user want to reset
          if (strcmp(usart_input_buffer, "reset") == 0) {
              //reset the player's chips to the initial state
              player_chips = (Chips) {
```

```
                                .yellow = 0, //$1000 chips
                                .purple = 1, //$500 chips
                                .black = 5, //$100 chips
                                .orange = 10, //$50 chips
                                .green = 12, //$25 chips
                                .blue = 10, //$10 chips
                                .red = 16, //$5 chips
                                .white = 20 //$1 chips
                            };
                            //update the chips and balance display
                            USART_print_chips(&player_chips, bet_amount);
                            //transition back to initial state
                            current_state = INIT_ST;
                        } else {
                            //invalid input
                            USART_ESC_Code(CLEAR_LINE);
                            USART_ESC_Code(FULLY_LEFT);
                            USART_print_string("Invalid input! Type 'reset' to
                                                start over.");

                            current_state = END_ST; //remain in end state
                        }
                    } else {
                        //inform the user that they can play again
                        USART_print_string("Press Enter to play again!");

                           while (!input_ready); //wait for user input
                           input_ready = false; //reset input flag

                        current_state = TRADE_ST; //transition to trade state
                    }
                     //turn off LEDs
                     GPIOB->ODR &= ~LED_PINS;
                     GPIOC->ODR &= ~LED_PINS;
                    break;
        }
    }
}

//ISR for USART2
void USART2_IRQHandler(void) {
    //check if RXNE flag is set
    if (USART2->ISR & USART_ISR_RXNE) {
        char c = USART2->RDR; //read received character
        if (c == '\b' || c == 127) { //handle backspace
          //move cursor back, print a space to 'erase', move back again
          if (usart_input_index > 0) {
                USART_ESC_Code(LEFT_1);
                USART_print_char(' ');
                USART_ESC_Code(LEFT_1);
                usart_input_index--; //remove last character from buffer
          }
```

```c
        } else if (c == '\n' || c == '\r') { //handle 'enter'
            //end of input
            usart_input_buffer[usart_input_index] = '\0'; //null-terminate string
            usart_input_index = 0; //reset buffer index
            input_ready = true;    //signal input is ready
        } else { //still typing
            //add character to buffer
            if (usart_input_index < sizeof(usart_input_buffer) - 1) {
                usart_input_buffer[usart_input_index++] = c;
                USART_print_char(c); //echo the character back
            }
        }
    }
}

//ISR for TIM2
void TIM2_IRQHandler(void) {
     //check if update flag is set
    if (TIM2->SR & TIM_SR_UIF) {
        TIM2->SR &= ~TIM_SR_UIF; //clear update flag
        if (!spin_complete) {
            //simulate wheel spinning
            spin_index = (spin_index + 1) % ARR_SIZE;
            USART_print_wheel(wheel_arr, spin_index);
            //alternate yellow and blue LEDs at visible rate
            if (spin_index % 9 == 0) {
               GPIOC->ODR ^= (YELLOW_PIN | BLUE_PIN);
            }
            //check if we completed the spin
            if (spin_iterations >= 5 && spin_index == winning_index) {
                spin_complete = true;
            }
            //increment iteration count if we completed a full spin
            if (spin_index == 0) {
                spin_iterations++;
            }
        }
    }
}

//handle bets that are stored in a single array
void handle_single_array_bet(const char **bet_array, uint8_t array_size) {
    //clear the winning numbers count
    winning_numbers_count = 0;
    //copy all numbers from the bet array to the winning_numbers array
    for (uint8_t i = 0; i < array_size; i++) {
        strncpy(winning_numbers[winning_numbers_count++], bet_array[i], 3);
    }

    current_state = TABLE_UPDATE_ST; //transition to table update state
}
```

```
//handle bets that are stored in a double array
void handle_double_array_bet(char *bet_name, const char **bet_array,
                              uint8_t row_size, uint8_t col_size) {
    //ask the user for the index of the row in the double array
    USART_ESC_Code(CLEAR_LINE);
    USART_ESC_Code(FULLY_LEFT);
    USART_print_string("Enter ");
    USART_print_string(bet_name);
    USART_print_string(" number (refer to user manual or table) --> ");

     while (!input_ready); //wait for user input
     input_ready = false; //reset input flag

    //validate the input
    uint8_t index = atoi(usart_input_buffer) - 1; //convert to 0-based index
    if (index >= 0 && index < row_size) {
        //clear the winning numbers count
        winning_numbers_count = 0;
        //access the correct row
        const char **row = bet_array + (index * col_size);
        //copy all numbers from the selected row to the winning_numbers array
        for (uint8_t i = 0; i < col_size; i++) {
            strncpy(winning_numbers[winning_numbers_count++], row[i], 3);
        }

        current_state = TABLE_UPDATE_ST; //transition to table update state
    } else {
     //invalid index
        USART_ESC_Code(CLEAR_LINE);
        USART_ESC_Code(FULLY_LEFT);
        USART_print_string("Invalid number! The number you entered does not exist
                            on the table.");
        HAL_Delay(DEL); //2 second delay

        current_state = BET_TYPE_ST; //remain in the betting type state
    }
}

//80MHz MCU clock, 48MHz RNG clock
void SystemClock_Config(void) {
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK) {
    Error_Handler();
  }
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
  RCC_OscInitStruct.MSIState = RCC_MSI_ON;
  RCC_OscInitStruct.MSICalibrationValue = 0;
  RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_6;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_MSI;
  RCC_OscInitStruct.PLL.PLLM = 1;
```

```
  RCC_OscInitStruct.PLL.PLLN = 40;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
    Error_Handler();
  }
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK) {
    Error_Handler();
  }
}
void Error_Handler(void) {
  __disable_irq();
  while (1){}
}
#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line) {}
#endif
```

misc.h

```
#ifndef SRC_MISC_H_
#define SRC_MISC_H_
#include "stm32l4xx_hal.h"
#include <string.h>

#define YELLOW_VAL 1000 //yellow chips are $1000
#define PURPLE_VAL 500 //purple chips are $500
#define BLACK_VAL 100 //black chips are $100
#define ORANGE_VAL 50 //orange chips are $50
#define GREEN_VAL 25 //green chips are $25
#define BLUE_VAL 10 //blue chips are $10
#define RED_VAL 5 //red chips are $5
#define WHITE_VAL 1 //white chips are $1
#define POSSIBLE_CHIPS 8 //number of different chips
#define LED_PINS (GPIO_ODR_OD5 | GPIO_ODR_OD6 | GPIO_ODR_OD7 | GPIO_ODR_OD8)
#define YELLOW_PIN GPIO_ODR_OD2
#define BLUE_PIN GPIO_ODR_OD3

typedef struct {
    uint32_t yellow; //number of $1000 chips
    uint32_t purple; //number of $500 chips
    uint32_t black; //number of $100 chips
    uint32_t orange; //number of $50 chips
    uint32_t green; //number of $25 chips
    uint32_t blue; //number of $10 chips
    uint32_t red; //number of $5 chips
    uint32_t white; //number of $1 chips
} Chips;

void TIM2_init(void);
void LED_init(void);
void RNG_init(void);
uint32_t RNG_get_random_number(void);
uint32_t calculate_total_balance(Chips);
uint32_t calculate_odds(const char *);
void distribute_chips(uint32_t, Chips *);
uint32_t *get_chip_pointer(uint32_t, Chips *);

#endif
```

---

misc.c

```
#include "misc.h"

#define RNG_MULT 24 //clock configuration multiplier
#define ARR_VAL 2105263 //full wheel spin in 1 second
```

```
//configure TIM2
void TIM2_init(void){
     //turn on TIM2 clock
     RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN;
     //set timer to count in up mode
     TIM2->CR1 &= ~TIM_CR1_DIR;
     //set ARR to 1s
     TIM2->ARR = ARR_VAL - 1;
     //enable TIM2 in NVIC
     NVIC->ISER[0] = (1 << TIM2_IRQn);
}

//configure LEDs to output, push-pull, very fast, no pull up/pull down resistor
void LED_init(void) {
     //turn on GPIOB and GPIOC clock
     RCC->AHB2ENR |= (RCC_AHB2ENR_GPIOBEN | RCC_AHB2ENR_GPIOCEN);
     //red LEDS
     GPIOB->MODER &= ~(GPIO_MODER_MODE5 | GPIO_MODER_MODE6 |
                       GPIO_MODER_MODE7 | GPIO_MODER_MODE8);
     GPIOB->MODER |= ((1 << GPIO_MODER_MODE5_Pos) | (1 << GPIO_MODER_MODE6_Pos) |
                     (1 << GPIO_MODER_MODE7_Pos) | (1 << GPIO_MODER_MODE8_Pos));
     GPIOB->OTYPER &= ~(GPIO_OTYPER_OT5 | GPIO_OTYPER_OT6 |
                       GPIO_OTYPER_OT7 | GPIO_OTYPER_OT8);
     GPIOB->OSPEEDR |= (GPIO_OSPEEDR_OSPEED5 | GPIO_OSPEEDR_OSPEED6 |
                       GPIO_OSPEEDR_OSPEED7 | GPIO_OSPEEDR_OSPEED8);
     GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPD5 | GPIO_PUPDR_PUPD6 |
                      GPIO_PUPDR_PUPD7 | GPIO_PUPDR_PUPD8);
     //green LEDS (and 1 yellow, 1 blue)
     GPIOC->MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3 |
                      GPIO_MODER_MODE5 | GPIO_MODER_MODE6 |
                       GPIO_MODER_MODE7 | GPIO_MODER_MODE8);
     GPIOC->MODER |= ((1 << GPIO_MODER_MODE2_Pos) | (1 << GPIO_MODER_MODE3_Pos) |
                      (1 << GPIO_MODER_MODE5_Pos) | (1 << GPIO_MODER_MODE6_Pos) |
                      (1 << GPIO_MODER_MODE7_Pos) | (1 << GPIO_MODER_MODE8_Pos));
     GPIOC->OTYPER &= ~(GPIO_OTYPER_OT2 | GPIO_OTYPER_OT3 |
                      GPIO_OTYPER_OT5 | GPIO_OTYPER_OT6 |
                      GPIO_OTYPER_OT7 | GPIO_OTYPER_OT8);
     GPIOC->OSPEEDR |= (GPIO_OSPEEDR_OSPEED2 | GPIO_OSPEEDR_OSPEED3 |
                       GPIO_OSPEEDR_OSPEED5 | GPIO_OSPEEDR_OSPEED6 |
                       GPIO_OSPEEDR_OSPEED7 | GPIO_OSPEEDR_OSPEED8);
     GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPD2 | GPIO_PUPDR_PUPD3 |
                      GPIO_PUPDR_PUPD5 | GPIO_PUPDR_PUPD6 |
                      GPIO_PUPDR_PUPD7 | GPIO_PUPDR_PUPD8);
     //setting all pins to start low (LEDs off)
     GPIOB->ODR &= ~LED_PINS;
     GPIOC->ODR &= ~(LED_PINS | YELLOW_PIN | BLUE_PIN);
}

//initialize the RNG with 48MHz clock
void RNG_init(void) {
     //enable PLLSAIQ with multiplier of 24
     RCC->CR &= ~RCC_CR_PLLSAI1ON;
```

```c
     RCC->PLLSAI1CFGR &= ~RCC_PLLSAI1CFGR_PLLSAI1N_Msk;
     RCC->PLLSAI1CFGR |= (RNG_MULT << RCC_PLLSAI1CFGR_PLLSAI1N_Pos);
     RCC->PLLSAI1CFGR |= RCC_PLLSAI1CFGR_PLLSAI1QEN;
     RCC->CR |= RCC_CR_PLLSAI1ON;
     //configure RNG to use PLLSAI1Q
    RCC->CCIPR &= ~RCC_CCIPR_CLK48SEL;
    RCC->CCIPR |= RCC_CCIPR_CLK48SEL_0;
    //enable RNG clock
    RCC->AHB2ENR |= RCC_AHB2ENR_RNGEN;
    //enable RNG
    RNG->CR |= RNG_CR_RNGEN;
}

//generate a random number
uint32_t RNG_get_random_number(void) {
     //check for error bits
     if (RNG->SR & (RNG_SR_SEIS | RNG_SR_CEIS)) {
          //clear error bits
         RNG->SR &= ~(RNG_SR_CEIS | RNG_SR_SEIS);
     }
    //wait for the data ready flag
    while (!(RNG->SR & RNG_SR_DRDY));
    //return the random number
    return RNG->DR;
}

//calculate total balance based on number of chips
uint32_t calculate_total_balance(Chips chips) {
    uint32_t total_balance = (chips.yellow * YELLOW_VAL) +
                             (chips.purple * PURPLE_VAL) +
                             (chips.black * BLACK_VAL) +
                             (chips.orange * ORANGE_VAL) +
                             (chips.green * GREEN_VAL) +
                             (chips.blue * BLUE_VAL) +
                             (chips.red * RED_VAL) +
                             (chips.white * WHITE_VAL);
    return total_balance;
}

//determine odds/payout based on bet type
uint32_t calculate_odds(const char *bet_type) {
    if (strcmp(bet_type, "Straight") == 0) return 36;
    if (strcmp(bet_type, "Split") == 0) return 18;
    if (strcmp(bet_type, "Street") == 0) return 12;
    if (strcmp(bet_type, "Basket") == 0) return 12;
    if (strcmp(bet_type, "Corner") == 0) return 9;
    if (strcmp(bet_type, "Top Line") == 0) return 7;
    if (strcmp(bet_type, "Double Street") == 0) return 6;
    if (strcmp(bet_type, "Dozen") == 0 || strcmp(bet_type, "Column") == 0)
          return 3;
    if (strcmp(bet_type, "Red") == 0 || strcmp(bet_type, "Black") == 0 ||
        strcmp(bet_type, "Odd") == 0 || strcmp(bet_type, "Even") == 0 ||
```

```
        strcmp(bet_type, "Low") == 0 || strcmp(bet_type, "High") == 0) return 2;
    return 0; //default odds
}

//distribute winnings into chips, starting with highest chip amount
void distribute_chips(uint32_t amount, Chips *chips) {
    uint32_t chip_values[] = {YELLOW_VAL, PURPLE_VAL, BLACK_VAL, ORANGE_VAL,
GREEN_VAL, BLUE_VAL, RED_VAL, WHITE_VAL};
    uint32_t *chip_counts[] = {&chips->yellow, &chips->purple, &chips->black,
&chips->orange, &chips->green, &chips->blue, &chips->red, &chips->white};
    //distribute through all chip values if possible
    for (uint8_t i = 0; i < POSSIBLE_CHIPS; i++) {
        while (amount >= chip_values[i]) {
            (*chip_counts[i])++;
            amount -= chip_values[i];
        }
    }
}

//validate and get chip pointer for a given chip value
uint32_t* get_chip_pointer(uint32_t chip_value, Chips *chips) {
    switch (chip_value) {
        case YELLOW_VAL: return &chips->yellow;
        case PURPLE_VAL: return &chips->purple;
        case BLACK_VAL: return &chips->black;
        case ORANGE_VAL: return &chips->orange;
        case GREEN_VAL: return &chips->green;
        case BLUE_VAL: return &chips->blue;
        case RED_VAL: return &chips->red;
        case WHITE_VAL: return &chips->white;
        default: return NULL;
    }
}
```

usart.h

```
#ifndef SRC_USART_H_
#define SRC_USART_H_
#include "spots.h"
#include "misc.h"
#include <stdio.h>

#define DOWN_35 "[35B"
#define LEFT_1 "[1D"
#define LEFT_2 "[2D"
#define TOP_LEFT "[H"
#define CLEAR_LINE "[2K"
#define FULLY_LEFT "[1G"
#define RESET_ATTRIBUTES "[0m"

void USART_init(void);
void USART_print_char(char);
void USART_print_string(char*);
void USART_ESC_Code(char*);
void USART_reset_screen(void);
void USART_start_screen(void);
void USART_print_table(Spot*);
Spot USART_print_wheel(const Spot*, uint32_t);
void USART_print_chips(Chips*, uint32_t);

#endif
```

---

usart.c

```
#include "usart.h"

#define USART_PORT GPIOA //USART port
#define USART_AF 7 //USART alternating function
#define SYSTEM_CLK_FREQ 80000000 //80MHz MCU clock
#define BAUD_RATE 115200 //baud rate
#define NVIC_MASK 0x1F //mask bottom 5 bits
#define WHEEL_OUTLINE 4 //number of lines in wheel outline
#define TABLE_OUTLINE 16 //number of lines in table outline
#define BOTTOM_OUTLINE 10 //number of lines in bottom container outline

//escape codes
#define ESC "\x1B"
#define HIDE_CURSOR "[?25l"
#define UNDERLINE "[4m"
#define BOLD "[1m"
#define CLEAR_SCREEN "[2J"

#define RIGHT_1 "[1C"
```

```c
#define RIGHT_2 "[2C"
#define RIGHT_3 "[3C"
#define RIGHT_5 "[5C"
#define RIGHT_6 "[6C"
#define RIGHT_14 "[14C"
#define RIGHT_31 "[31C"
#define RIGHT_50 "[50C"
#define RIGHT_60 "[60C"
#define RIGHT_63 "[63C"

#define LEFT_6 "[6D"
#define LEFT_8 "[8D"

#define DOWN_1 "[1B"
#define DOWN_2 "[2B"
#define DOWN_5 "[5B"
#define DOWN_12 "[12B"
#define DOWN_23 "[23B"
#define DOWN_32 "[32B"

#define UP_9 "[9A"

#define YELLOW "[33m"
#define PURPLE "[35m"
#define BLACK "[30m"
#define ORANGE "[38:5:202m"
#define GREEN "[32m"
#define BLUE "[34m"
#define RED "[31m"
#define CYAN "[96m"

const char *wheel_outline[] = { //outline for wheel
    "####",
    "          -------------------|----|-------------------",
    "              |    |    |    |    |    |    |    |    ",
    "          -------------------|----|-------------------\n\n"
};

const char *table_outline[] = { //outline for table
    "   -------------------------------------------------------------",
    "   |            1 - 18           |           19 - 36           |",
    "---|-----------------------------|-----------------------------|------",
    "|  |    |    |    |    |    |    |    |    |    |    |    |    | 3rd |",
    "|  |----|----|----|----|----|----|----|----|----|----|----|----|-----|",
    "|--|    |    |    |    |    |    |    |    |    |    |    |    | 2nd |",
    "|  |----|----|----|----|----|----|----|----|----|----|----|----|-----|",
    "|  |    |    |    |    |    |    |    |    |    |    |    |    | 1st |",
    "---|-------------------|-------------------|-------------------|------",
    "   |       1 - 12      |       13 - 24      |       25 - 36      |",
    "   |-----------------------------------------------------------|",
    "   |     EVEN     |     RED      |    BLACK     |     ODD      |",
    "   -------------------------------------------------------------",
```

```
    "                                                         ---------------",
    "                                                         |BET: $       |",
    "                                                         ---------------"
};

const char *bottom_container_outline[] = { //outline for bottom container
    "---------------------------------------------    ----------------------",
    "| Straight: 35 to 1 | Double Street: 5 to 1 |    |     :   |   :      |",
    "| Split: 17 to 1    | Dozen: 2 to 1         |    |    :    |   :      |",
    "| Street: 11 to 1   | Column: 2 to 1        |    |    :    |  :       |",
    "| Basket: 11 to 1   | Red/Black: 1 to 1     |    |   :     |$1:       |",
    "| Corner: 8 to 1    | Odd/Even: 1 to 1      |    ----------------------",
    "| Top Line: 6 to 1  | Low/High: 1 to 1      |    |BALANCE: $          |",
    "---------------------------------------------    ----------------------",
    "-----------------------------------------------------------------------\n",
    "-----------------------------------------------------------------------"
};

//configure USART registers and pins
void USART_init(void) {
     //enable GPIOA clock
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
     //configure GPIO pins
    //set PA2 and PA3 to AF, push-pull, very fast, no PUPD resistor
    USART_PORT->MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);
    USART_PORT->MODER |= (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1);
    USART_PORT->OTYPER &= ~(GPIO_OTYPER_OT2 | GPIO_OTYPER_OT3);
    USART_PORT->OSPEEDR |= (GPIO_OSPEEDR_OSPEED2 | GPIO_OSPEEDR_OSPEED3);
    USART_PORT->PUPDR &= ~(GPIO_PUPDR_PUPD2 | GPIO_PUPDR_PUPD3);
    //clear and set USART pins to alternate function 7 (USART2)
    USART_PORT->AFR[0] &= ~(GPIO_AFRL_AFSEL2 | GPIO_AFRL_AFSEL3);
    USART_PORT->AFR[0] |= (USART_AF << GPIO_AFRL_AFSEL2_Pos |
                           USART_AF << GPIO_AFRL_AFSEL3_Pos);
     //enable USART2 clock
     RCC->APB1ENR1 |= RCC_APB1ENR1_USART2EN;
     //set word length to 1 start bit, 8 data bits, 1 stop bit, no parity bit
     USART2->CR1 &= ~(USART_CR1_M0 | USART_CR1_M1);
     USART2->CR2 &= ~USART_CR2_STOP;
     USART2->CR1 &= ~USART_CR1_PCE;
     //set baud rate (oversampling by 16)
     USART2->CR1 &= ~USART_CR1_OVER8;
     USART2->BRR = (uint32_t)(SYSTEM_CLK_FREQ/BAUD_RATE);
     //set TE bit to send an idle frame as first transmission
     //enable receiver and receiver interrupts
     USART2->CR1 |= (USART_CR1_TE | USART_CR1_RE | USART_CR1_RXNEIE);
     NVIC->ISER[1] |= (1 << (USART2_IRQn & NVIC_MASK));
     //enable USART
     USART2->CR1 |= USART_CR1_UE;
}

//transmit character
void USART_print_char(char input) {
```

```
        //wait until TXE is set
        while (!(USART2->ISR & USART_ISR_TXE));
        //transmit character
        USART2->TDR = input;
}

//transmit a string of characters
void USART_print_string(char* input) {
        //continue to transmit characters until reaching end of string
       while (*input != '\0') {
        USART_print_char(*input);
            input++;
       }
}

//print ESC character, then print desired ESC code
void USART_ESC_Code(char* code) {
        USART_print_string(ESC);
        USART_print_string(code);
}

//reset terminal screen
void USART_reset_screen(void) {
        USART_ESC_Code(CLEAR_SCREEN);
        USART_ESC_Code(TOP_LEFT);
        USART_ESC_Code(RESET_ATTRIBUTES);
        USART_ESC_Code(HIDE_CURSOR);
}

//print line (of table/wheel outline)
void USART_print_line(char* line) {
        USART_print_string(line);
        USART_ESC_Code(DOWN_1);
        USART_ESC_Code(FULLY_LEFT);
}

//print inputted section line by line
void USART_print_section(const char **lines, uint8_t count) {
       for (uint8_t i = 0; i < count; i++) {
            USART_print_line(lines[i]);
       }
}

//print start screen
void USART_start_screen(void) {
        //reset screen
        USART_reset_screen();
       //print title
       USART_ESC_Code(RIGHT_31);
       USART_ESC_Code(BOLD);
       USART_ESC_Code(UNDERLINE);
       USART_print_string("ROULETTE\n\n\n");
```

```
    USART_ESC_Code(RESET_ATTRIBUTES);
    USART_ESC_Code(LEFT_6);
    //print wheel outline
    USART_print_section(wheel_outline, WHEEL_OUTLINE);
    //print table outline
    USART_print_section(table_outline, TABLE_OUTLINE);
    //print bottom container outline
    USART_ESC_Code(BOLD);
    USART_print_string("             BETTING PAYOUTS
                   CHIPS\n");
    USART_ESC_Code(FULLY_LEFT);
    USART_ESC_Code(RESET_ATTRIBUTES);
    USART_print_section(bottom_container_outline, BOTTOM_OUTLINE);
     //print colored chip values
     USART_ESC_Code(UP_9);
     USART_ESC_Code(FULLY_LEFT);
     USART_ESC_Code(RIGHT_50);
     USART_ESC_Code(YELLOW);
     USART_print_string("$1000");
     USART_ESC_Code(GREEN);
     USART_ESC_Code(RIGHT_5);
     USART_print_string("$25\n");

     USART_ESC_Code(FULLY_LEFT);
     USART_ESC_Code(RIGHT_50);
     USART_ESC_Code(PURPLE);
     USART_print_string("$500");
     USART_ESC_Code(BLUE);
     USART_ESC_Code(RIGHT_6);
     USART_print_string("$10\n");

     USART_ESC_Code(FULLY_LEFT);
     USART_ESC_Code(RIGHT_50);
     USART_ESC_Code(BLACK);
     USART_print_string("$100");
     USART_ESC_Code(RED);
     USART_ESC_Code(RIGHT_6);
     USART_print_string("$5\n");

     USART_ESC_Code(FULLY_LEFT);
     USART_ESC_Code(RIGHT_50);
     USART_ESC_Code(ORANGE);
     USART_print_string("$50");
     USART_ESC_Code(RESET_ATTRIBUTES);
}

//print given table spots in correct locations with appropriate colors
void USART_print_table(Spot *table_arr) {
     USART_ESC_Code(TOP_LEFT);
     USART_ESC_Code(DOWN_12);
     USART_ESC_Code(RIGHT_1);
     USART_ESC_Code(BOLD);
```

```
    //print "00"
    if (strcmp(table_arr[0].color, "cyan") == 0) {
        USART_ESC_Code(CYAN);
    } else {
        USART_ESC_Code(GREEN);
    }
     USART_print_string(table_arr[0].number);
     USART_ESC_Code(RIGHT_2);
    //print columns
    for (uint8_t i = 2; i <= ARR_SIZE - 1; i++) {
     //choose color
        if (strcmp(table_arr[i].color, "red") == 0) {
            USART_ESC_Code(RED);
        } else if (strcmp(table_arr[i].color, "black") == 0) {
          USART_ESC_Code(BLACK);
        } else if (strcmp(table_arr[i].color, "cyan") == 0) {
            USART_ESC_Code(CYAN);
        }
        USART_print_string(table_arr[i].number);
     USART_ESC_Code(RIGHT_3);
     //adjust at end of row
     if (i == 13 || i == 25) {
          USART_ESC_Code(FULLY_LEFT);
          USART_ESC_Code(DOWN_2);
          USART_ESC_Code(RIGHT_5);
     }
    }
     USART_ESC_Code(FULLY_LEFT);
     USART_ESC_Code(RIGHT_1);
     //print "0"
    if (strcmp(table_arr[1].color, "cyan") == 0) {
        USART_ESC_Code(CYAN);
    } else {
        USART_ESC_Code(GREEN);
    }
     USART_print_string(table_arr[1].number);
     USART_ESC_Code(RESET_ATTRIBUTES);
}

//print wheel in wheel outline
Spot USART_print_wheel(const Spot *wheel_arr, uint32_t index) {
     //number of spots to display in the row
    const uint8_t display_count = 9;
    const uint8_t half_window = display_count / 2;
    //calculate the starting index for the circular array
    uint8_t start_index = (index - half_window + ARR_SIZE) % ARR_SIZE;
    //navigate to outline
    USART_ESC_Code(TOP_LEFT);
    USART_ESC_Code(DOWN_5);
    USART_ESC_Code(RIGHT_14);
    USART_ESC_Code(BOLD);
    //variable to hold middle "winning" spot
```

```
    Spot winning_spot = wheel_arr[(start_index + half_window) % ARR_SIZE];
    //print the 9 spots
    for (uint8_t i = 0; i < display_count; i++) {
        //compute the current index in the circular array
     uint8_t current_index = (start_index + i) % ARR_SIZE;
        //set color for the spot
        if (strcmp(wheel_arr[current_index].color, "red") == 0) {
          USART_ESC_Code(RED);
        } else if (strcmp(wheel_arr[current_index].color, "black") == 0) {
          USART_ESC_Code(BLACK);
        } else if (strcmp(wheel_arr[current_index].color, "green") == 0) {
          USART_ESC_Code(GREEN);
        }
        //print the number
        USART_print_string(wheel_arr[current_index].number);
        USART_ESC_Code(RIGHT_3);
    }
    return winning_spot; //return winning spot
}

//print inputted balance
void USART_print_balance(uint32_t balance) {
    char balance_str[12]; //character buffer
    //navigate to balance position
    USART_ESC_Code(TOP_LEFT);
    USART_ESC_Code(RESET_ATTRIBUTES);
    USART_ESC_Code(DOWN_32);
    USART_ESC_Code(RIGHT_60);
    //clear spots by overwriting with spaces
    for (uint8_t i = 0; i < 8; i++) {
        USART_print_char(' ');
    }
    //move cursor back to the start of the cleared area
    USART_ESC_Code(LEFT_8);
    //convert balance to string and print
    snprintf(balance_str, sizeof(balance_str), "%lu", balance);
    USART_print_string(balance_str);
}

//print inputted bet
void USART_print_bet(uint32_t bet) {
    char bet_str[12]; //character buffer
    //naviagte to bet position
    USART_ESC_Code(TOP_LEFT);
    USART_ESC_Code(RESET_ATTRIBUTES);
    USART_ESC_Code(DOWN_23);
    USART_ESC_Code(RIGHT_63);
    //clear spots by overwriting with spaces
    for (uint8_t i = 0; i < 6; i++) {
        USART_print_char(' ');
    }
    //move cursor back to the start of the cleared area
```

```
    USART_ESC_Code(LEFT_6);
    //convert bet to string and print
    snprintf(bet_str, sizeof(bet_str), "%lu", bet);
    USART_print_string(bet_str);
}

//print inputted chips
void USART_print_chips(Chips *chips, uint32_t bet) {
    char chip_count_str[4]; //buffer to store chip counts
    USART_ESC_Code(RESET_ATTRIBUTES);
    //clear the area and print chip values
    void USART_clear_and_print(uint8_t down, uint8_t right, uint32_t value) {
        //navigate to position
        USART_ESC_Code(TOP_LEFT);
        for (uint8_t i = 0; i < down; i++) {
            USART_ESC_Code(DOWN_1);
        }
        for (uint8_t i = 0; i < right; i++) {
            USART_ESC_Code(RIGHT_1);
        }
        //clear area (overwrite with spaces)
        for (uint8_t i = 0; i < 3; i++) {
            USART_print_char(' ');
        }
        //move back to the start of the cleared area
        for (uint8_t i = 0; i < 3; i++) {
            USART_ESC_Code(LEFT_1);
        }
        //print the value
        snprintf(chip_count_str, sizeof(chip_count_str), "%lu", value);
        USART_print_string(chip_count_str);
    }
    //print chips in corresponding spots
    USART_clear_and_print(27, 56, chips->yellow); //$1000 chips
    USART_clear_and_print(28, 55, chips->purple); //$500 chips
    USART_clear_and_print(29, 55, chips->black); //$100 chips
    USART_clear_and_print(30, 54, chips->orange); //$50 chips
    USART_clear_and_print(27, 64, chips->green); //$25 chips
    USART_clear_and_print(28, 64, chips->blue); //$10 chips
    USART_clear_and_print(29, 63, chips->red); //$5 chips
    USART_clear_and_print(30, 63, chips->white); //$1 chips
    //update total balance and bet
    uint32_t total_balance = calculate_total_balance(*chips);
    USART_print_bet(bet);
    USART_print_balance(total_balance);
}
```

spots.h

```
#ifndef SRC_SPOTS_H_
#define SRC_SPOTS_H_
#include "stm32l4xx_hal.h"

#define ARR_SIZE 38 //total spots
#define SPLIT_SIZE 2 //number of spots in a split
#define ST_SIZE 3 //number of spots in a street
#define BASKET_SIZE 3 //number of spots in a basket
#define CORNER_SIZE 4 //number of spots in a corner
#define DUB_ST_SIZE 6 //number of spots in a double street
#define DOZ_COL_SIZE 12 //number of spots in a dozen/column

//structure to represent spot
typedef struct {
    char color[6]; //spot color
    char number[3]; //spot number
} Spot;

extern const Spot wheel_arr[ARR_SIZE];
extern const Spot base_table_arr[ARR_SIZE];
extern const char *split_bets[][SPLIT_SIZE];
extern const char *street_bets[][ST_SIZE];
extern const char *basket_bets[][BASKET_SIZE];
extern const char *corner_bets[][CORNER_SIZE];
extern const char *top_line[];
extern const char *double_street_bets[][DUB_ST_SIZE];
extern const char *dozen_bets[][DOZ_COL_SIZE];
extern const char *column_bets[][DOZ_COL_SIZE];
extern const char *red[];
extern const char *black[];
extern const char *odds[];
extern const char *evens[];
extern const char *low_half[];
extern const char *high_half[];

#endif
```

---

spots.c

```
#include "spots.h"

//base colored table spots
const Spot base_table_arr[ARR_SIZE] = {
     {"green", "00"},{"green", " 0"},{"red", " 3"},{"black", " 6"},{"red", " 9"},
     {"red", "12"}, {"black", "15"}, {"red", "18"},{"red", "21"},{"black", "24"},
     {"red", "27"}, {"red", "30"}, {"black", "33"},{"red", "36"},{"black", " 2"},
     {"red", " 5"},{"black", " 8"},{"black", "11"},{"red", "14"},{"black", "17"},
```

```
    {"black", "20"},{"red", "23"},{"black", "26"},{"black", "29"},{"red", "32"},
    {"black", "35"},{"red", " 1"},{"black", " 4"},{"red", " 7"},{"black", "10"},
    {"black", "13"}, {"red", "16"}, {"red", "19"},{"black", "22"},{"red", "25"},
    {"black", "28"}, {"black", "31"}, {"red", "34"}
};
//wheel spots
const Spot wheel_arr[ARR_SIZE] = {
    {"green", "00"},{"red", "27"},{"black", "10"},{"red", "25"},{"black", "29"},
    {"red", "12"}, {"black", " 8"}, {"red", "19"},{"black", "31"},{"red", "18"},
    {"black", " 6"},{"red", "21"},{"black", "33"},{"red", "16"},{"black", " 4"},
    {"red", "23"},{"black", "35"},{"red", "14"},{"black", " 2"},{"green", " 0"},
    {"black", "28"},{"red", " 9"},{"black", "26"},{"red", "30"},{"black", "11"},
    {"red", " 7"}, {"black", "20"}, {"red", "32"},{"black", "17"},{"red", " 5"},
    {"black", "22"},{"red", "34"},{"black", "15"},{"red", " 3"},{"black", "24"},
    {"red", "36"}, {"black", "13"}, {"red", " 1"}
};
//possible split bets
const char *split_bets[][SPLIT_SIZE] = {
   {"0", "1"}, {"0", "2"}, {"00", "2"}, {"00", "3"}, {"1", "2"}, {"2", "3"},
   {"1", "4"}, {"2", "5"}, {"3", "6"}, {"4", "5"}, {"5", "6"}, {"4", "7"},
   {"5", "8"}, {"6", "9"}, {"7", "8"}, {"8", "9"}, {"7", "10"}, {"8", "11"},
   {"9", "12"},{"10", "11"},{"11", "12"},{"10", "13"},{"11", "14"},{"12", "15"},
   {"13", "14"},{"14", "15"},{"13", "16"},{"14", "17"},{"15", "18"},{"16","17"},
   {"17", "18"},{"16", "19"},{"17", "20"},{"18", "21"},{"19", "20"},{"20","21"},
   {"19", "22"},{"20", "23"},{"21", "24"},{"22", "23"},{"23", "24"},{"22","25"},
   {"23", "26"},{"24", "27"},{"25", "26"},{"26", "27"},{"25", "28"},{"26","29"},
   {"27", "30"},{"28", "29"},{"29", "30"},{"28", "31"},{"29", "32"},{"30","33"},
   {"31", "32"},{"32", "33"},{"31", "34"},{"32", "35"},{"33", "36"},{"34","35"},
   {"35", "36"}
};
//possible streets
const char *street_bets[][ST_SIZE] = {
   {"1", "2", "3"}, {"4", "5", "6"}, {"7", "8", "9"}, {"10", "11", "12"},
    {"13", "14", "15"},{"16", "17", "18"},{"19", "20", "21"},{"22", "23", "24"},
   {"25", "26", "27"}, {"28", "29", "30"}, {"31", "32", "33"},{"34", "35", "36"}
};
//possible baskets
const char *basket_bets[][BASKET_SIZE] = {
   {"0", "1", "2"}, {"0", "00", "2"}, {"00", "2", "3"}
};
//possible corners
const char *corner_bets[][CORNER_SIZE] = {
   {"1", "2","4","5"},{"2", "3","5","6"},{"4", "5","7","8"},{"5", "6","8", "9"},
   {"7", "8", "10", "11"}, {"8", "9", "11", "12"}, {"10", "11", "13", "14"},
   {"11", "12", "14", "15"}, {"13", "14", "16", "17"}, {"14", "15", "17", "18"},
   {"16", "17", "19", "20"}, {"17", "18", "20", "21"}, {"19", "20", "22", "23"},
   {"20", "21", "23", "24"}, {"22", "23", "25", "26"}, {"23", "24", "26", "27"},
   {"25", "26", "28", "29"}, {"26", "27", "29", "30"}, {"28", "29", "31", "32"},
   {"29", "30", "32", "33"}, {"31", "32", "34", "35"}, {"32", "33", "35", "36"}
};
//top line
const char *top_line[] = {"0", "00", "1", "2", "3"};
```

```
//possible double streets
const char *double_street_bets[][DUB_ST_SIZE] = {
    {"1", "2", "3", "4", "5", "6"}, {"4", "5", "6", "7", "8", "9"},
    {"7", "8", "9", "10", "11", "12"}, {"10", "11", "12", "13", "14", "15"},
    {"13", "14", "15", "16", "17", "18"}, {"16", "17", "18", "19", "20", "21"},
    {"19", "20", "21", "22", "23", "24"}, {"22", "23", "24", "25", "26", "27"},
    {"25", "26", "27", "28", "29", "30"}, {"28", "29", "30", "31", "32", "33"},
    {"31", "32", "33", "34", "35", "36"}
};
//possible dozens
const char *dozen_bets[][DOZ_COL_SIZE] = {
    {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12"},
    {"13", "14", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24"},
    {"25", "26", "27", "28", "29", "30", "31", "32", "33", "34", "35", "36"}
};
//possible columns
const char *column_bets[][DOZ_COL_SIZE] = {
    {"1", "4", "7", "10", "13", "16", "19", "22", "25", "28", "31", "34"},
    {"2", "5", "8", "11", "14", "17", "20", "23", "26", "29", "32", "35"},
    {"3", "6", "9", "12", "15", "18", "21", "24", "27", "30", "33", "36"}
};
//possible reds
const char *red[] = {
    "1", "3", "5", "7", "9", "12", "14", "16", "18",
    "19", "21", "23", "25", "27", "30", "32", "34", "36"
};
//possible blacks
const char *black[] = {
    "2", "4", "6", "8", "10", "11", "13", "15", "17",
    "20", "22", "24", "26", "28", "29", "31", "33", "35"
};
//possible odds
const char *odds[] = {
    "1", "3", "5", "7", "9", "11", "13", "15", "17",
    "19", "21", "23", "25", "27", "29", "31", "33", "35"
};
//possible evens
const char *evens[] = {
    "2", "4", "6", "8", "10", "12", "14", "16", "18",
    "20", "22", "24", "26", "28", "30", "32", "34", "36"
};
//possible numbers from lower half
const char *low_half[] = {
    "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
    "11", "12", "13", "14", "15", "16", "17", "18"
};
//possible numbers from upper half
const char *high_half[] = {
    "19", "20", "21", "22", "23", "24", "25", "26", "27", "28",
    "29", "30", "31", "32", "33", "34", "35", "36"
};
```

REFERENCES

[1] STMicroelectronics NV, "RM0351 Reference manual, STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs," [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. [Accessed 11 December 2024].

[2] STMicroelectronics NV, "STM32L476xx Data Sheet," [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf. [Accessed 11 December 2024].