Embedded Pong on ST7735 TFT with SPI Graphics and ESP32 Wireless Control

Krishna Karthikeya Chemudupati

1 Overview

This project implements a complete Pong game on the ATmega328PB microcontroller, rendered on an ST7735 TFT LCD display via SPI. The system supports dual-input control: an analog joystick connected through the ADC and wireless paddle control via an ESP32 running the Blynk IoT platform. Peripheral feedback includes a PWM buzzer for audio events, RGB LEDs for score indication, and a motor for victory haptics.

The game features ball physics with velocity tracking, paddle collision detection, a computer-controlled AI opponent, and score tracking with a configurable win condition (first to 2 points).

Table 1: Hardware Components

Component	**Role**
ATmega328PB Xplained Mini	Main microcontroller (16 MHz)
ST7735 TFT LCD	160×128 pixel display, RGB565 color
Analog Joystick	Player paddle control (ADC channels 4–5)
ESP32	Wireless Blynk app control
Buzzer	PWM audio feedback
2× LEDs (Green, Red)	Score indication
DC Motor	Victory haptic feedback

2 System Architecture

2.1 SPI Communication with the ST7735 LCD

The ATmega328PB communicates with the ST7735 LCD controller over SPI. The interface uses four signals, all driven from the MCU to the LCD. The serial clock SCK on PB5 synchronizes transfers, with data latched on the rising edge consistent with SPI Mode 0. Chip select CS on PB2 is an active-low enable—the LCD only accepts data while CS is asserted low. The serial data line SDI (MOSI) on PB3 carries both commands and pixel data from the MCU to the LCD. Finally, the data/command pin D/CX on PB0 differentiates between command bytes (D/CX low) and data bytes (D/CX high), allowing the controller to determine whether an incoming byte configures the display or writes to pixel memory.

The SPI peripheral is initialized at `fck/64` with SPI2X for doubled speed:

```c
void SPI_Controller_Init(void) {
    DDRB |= (1 << LCD_MOSI) | (1 << LCD_SCK)
          | (1 << LCD_TFT_CS) | (1 << LCD_DC);
    SPCR |= (1 << SPE) | (1 << MSTR) | (1 << SPR1);
    SPSR |= (1 << SPI2X);
}
```

Listing 1: SPI controller initialization

2.2 RGB565 Color Encoding

The display uses 16-bit RGB565 color, where Red and Blue each have 5 bits (32 levels) and Green has 6 bits (64 levels), reflecting the human eye's greater sensitivity to green. Each 16-bit color value is transmitted as two sequential 8-bit SPI transfers:

```c
void SPI_ControllerTx_16bit(uint16_t data) {
    SPI_ControllerTx(data >> 8);     // High byte: R[4:0], G[5:3]
    SPI_ControllerTx(data & 0xFF);   // Low byte:  G[2:0], B[4:0]
}
```

Listing 2: 16-bit color transmission over 8-bit SPI

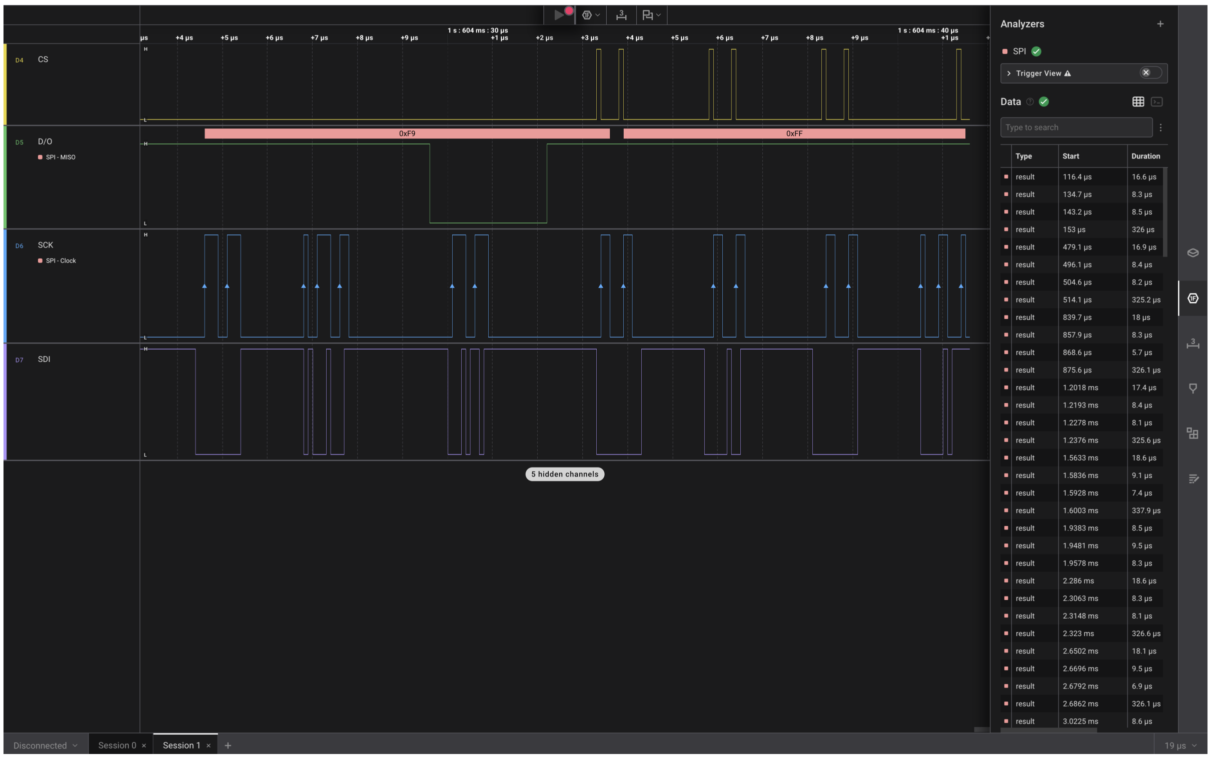
The `rgb565()` conversion function maps 24-bit RGB888 values to 16-bit RGB565. While the conversion loses color precision (65,536 vs. 16.7M colors), the difference is not visually significant for embedded LCD applications and halves the required SPI bandwidth.

2.3 LCD Address Windowing

The `LCD_setAddr()` function defines a rectangular region in the LCD's display memory for subsequent pixel writes. It first sends `ST7735_CASET` (0x2A) to set the column address range, then `ST7735_RASET` (0x2B) for the row address range, and finally `ST7735_RAMWR` (0x2C) to initiate RAM write mode. This windowing mechanism allows efficient block fills and partial screen updates without redrawing the entire display.

3 SPI Signal Analysis

The SPI communication was verified using a Saleae Logic Analyzer. Figure 1 shows a captured transaction with all four signals labeled.



	Type	Start	Duration
🟥	result	116.4 µs	16.6 µs
🟥	result	134.7 µs	8.3 µs
🟥	result	143.2 µs	8.5 µs
🟥	result	153 µs	326 µs
🟥	result	479.1 µs	16.9 µs
🟥	result	496.1 µs	8.4 µs
🟥	result	504.6 µs	8.2 µs
🟥	result	514.1 µs	325.2 µs
🟥	result	839.7 µs	18 µs
🟥	result	857.9 µs	8.3 µs
🟥	result	868.6 µs	5.7 µs
🟥	result	875.6 µs	326.1 µs
🟥	result	1.2018 ms	17.4 µs
🟥	result	1.2193 ms	8.4 µs
🟥	result	1.2278 ms	8.1 µs
🟥	result	1.2376 ms	325.6 µs
🟥	result	1.5633 ms	18.6 µs
🟥	result	1.5836 ms	9.1 µs
🟥	result	1.5928 ms	7.4 µs
🟥	result	1.6003 ms	337.9 µs
🟥	result	1.9383 ms	8.5 µs
🟥	result	1.9481 ms	9.5 µs
🟥	result	1.9578 ms	8.3 µs
🟥	result	2.286 ms	18.6 µs
🟥	result	2.3063 ms	8.3 µs
🟥	result	2.3148 ms	8.1 µs
🟥	result	2.323 ms	326.6 µs
🟥	result	2.6502 ms	18.1 µs
🟥	result	2.6696 ms	9.5 µs
🟥	result	2.6792 ms	6.9 µs
🟥	result	2.6862 ms	326.1 µs
🟥	result	3.0225 ms	8.6 µs

Figure 1: Saleae Logic Analyzer capture of SPI signals: CS, D/CX, SCK, and SDI during an LCD transaction. The Analyzers panel on the right shows decoded SPI results with timing data.

The SDI line holds a stable logic level during the low period of SCK and transitions only between clock pulses. Data is latched on the rising edge of SCK, consistent with SPI Mode 0. CS is asserted low for approximately 3.2 μs per transaction, and within each CS-low window 8 SCK pulses are observed, corresponding to 1 byte of data on SDI. The D/CX signal clearly toggles between command phases (low) and data phases (high), showing distinct command–data sequencing in the protocol.

4 Graphics Library

A custom `LCD_GFX` graphics library provides drawing primitives on top of the SPI LCD driver. The library implements the following functions:

Table 2: LCD_GFX Drawing Primitives

Function	Description
`LCD_drawPixel()`	Draws a single RGB565 pixel at (x, y)
`LCD_drawChar()`	Renders one ASCII character using a 5×8 bitmap lookup table
`LCD_drawString()`	Renders a null-terminated string with 6-pixel character spacing
`LCD_drawCircle()`	Midpoint circle algorithm with 8-way symmetry
`LCD_drawLine()`	Bresenham's line algorithm
`LCD_drawBlock()`	Fills a rectangular region using address windowing
`LCD_setScreen()`	Fills the entire 160×128 display with a single color
`rgb565()`	Converts RGB888 to RGB565 format

Predefined RGB565 color constants are provided: `BLACK` (0x0000), `WHITE` (0xFFFF), `RED` (0xF800), `GREEN` (0x07E0), `BLUE` (0x001F), `CYAN` (0x07FF), `MAGENTA` (0xF81F), and `YELLOW` (0xFFE0).

The circle drawing uses the midpoint algorithm, which exploits 8-way symmetry to minimize per-pixel computation:

```c
void LCD_drawCircle(uint8_t x0, uint8_t y0,
                    uint8_t radius, uint16_t color) {
    int xPos = radius, yPos = 0, error = 0;
    while (xPos >= yPos) {
        // Draw 8 symmetrical points
        LCD_drawPixel(x0 + xPos, y0 + yPos, color);
        LCD_drawPixel(x0 - xPos, y0 - yPos, color);
        // ... (6 more octant points)
        if (error <= 0) { yPos++; error += 2*yPos + 1; }
        else            { xPos--; error -= 2*xPos + 1; }
    }
}
```

Listing 3: Midpoint circle algorithm (excerpt)

5 Game Implementation

5.1 Data Structures

The game state is organized into three structs:

```c
typedef struct {
    int16_t x, y;     // Position
    int8_t  vx, vy;   // Velocity components
} Ball;

typedef struct {
    int16_t y;          // Vertical position
    uint8_t score;
} Paddle;

typedef struct {
    Ball   ball;
    Paddle player;
    Paddle computer;
    bool    game_active;
} GameState;
```

Listing 4: Game data structures

5.2 Game Constants

Table 3: Game Parameters

Parameter	**Macro**	**Value**
Screen resolution	–	160×128 px
Paddle dimensions	`PADDLE_WIDTH/HEIGHT`	4×20 px
Ball radius	`BALL_RADIUS`	3 px
Paddle speed	`PADDLE_SPEED`	3 px/frame
Winning score	`WINNING_SCORE`	2 points
Frame rate	`_delay_ms(20)`	∼50 FPS

5.3 Ball Physics and Collision Detection

The ball updates its position each frame by adding velocity components to coordinates. Wall collisions (top/bottom boundaries) reverse the vertical velocity. Paddle collisions check whether the ball overlaps with either paddle's bounding box and reverse the horizontal velocity accordingly, repositioning the ball just outside the paddle to prevent tunneling:

```
// Player paddle collision
if (game.ball.x - BALL_RADIUS <= PADDLE_WIDTH
    && game.ball.vx < 0
    && game.ball.y >= game.player.y
    && game.ball.y <= game.player.y + PADDLE_HEIGHT) {
    game.ball.vx = -game.ball.vx;
    game.ball.x = PADDLE_WIDTH + BALL_RADIUS + 1;
    play_tone(3000, 50);
}
```

Listing 5: Paddle collision detection

5.4 Computer AI Opponent

The AI paddle tracks the ball's vertical position with intentional delay—it only updates every 10th game tick using a static counter. This creates beatable difficulty while maintaining responsive gameplay:

```
void update_computer_paddle(void) {
    static uint16_t ai_counter = 0;
    ai_counter++;
    if (ai_counter % 10 != 0) return;  // Update every 10th tick

    int16_t ball_center   = game.ball.y;
    int16_t paddle_center = game.computer.y + PADDLE_HEIGHT / 2;

    if (ball_center < paddle_center - 2)
        game.computer.y -= PADDLE_SPEED;
    else if (ball_center > paddle_center + 2)
        game.computer.y += PADDLE_SPEED;
}
```

Listing 6: AI opponent logic

5.5 Scoring and Win Conditions

When the ball passes a paddle (exits the left or right screen boundary), the opposing player scores. The corresponding LED flashes (green for player, red for computer) and a tone plays. At the winning score threshold, a victory message is displayed, the motor spins as haptic feedback, and the game resets after a 3-second delay.

After each point, the ball respawns at screen center with a randomized velocity direction.

6 Input Control

6.1 Analog Joystick

The joystick is connected to ADC channels 4 and 5 (PC4, PC5). The ADC is configured with AVCC reference and a prescaler of 128 for accurate 10-bit conversion at 16 MHz:

```
ADMUX  = (1 << REFS0);           // AVCC reference
ADCSRA = (1 << ADEN) | (1 << ADPS2)
       | (1 << ADPS1) | (1 << ADPS0);  // Prescaler 128
```

Listing 7: ADC configuration

Joystick readings are thresholded into three zones: below 300 (up), above 700 (down), or neutral (no movement).

6.2 ESP32 Wireless Control via Blynk

The ESP32 runs an Arduino sketch that connects to the Blynk IoT platform over WiFi. Two virtual buttons (V0, V1) in the Blynk mobile app control GPIO17 and GPIO18 on the ESP32, which are wired to PD0 and PD1 on the ATmega328PB as active-low inputs with internal pull-ups.

```
BLYNK_WRITE(V0) {
    if (param.asInt() == 1)
        digitalWrite(18, HIGH);  // Signal ATmega328PB
    else
        digitalWrite(18, LOW);
}
```

Listing 8: Blynk virtual pin handler on ESP32

The ATmega328PB polls these pins each frame with a 5-iteration smoothing loop (2 ms per iteration). Wireless control takes priority over joystick input—if the ESP32 signals movement, the joystick reading is skipped for that frame.

Button debouncing is unnecessary because the Blynk virtual buttons are software-controlled and produce clean digital signals without mechanical bounce.

7 Peripheral Feedback

7.1 Audio (Buzzer)

The buzzer on PD3 generates tones using software-driven PWM at approximately 2% duty cycle. Different frequencies signal different events: 2500 Hz for 50 ms on wall bounces, 3000 Hz for 50 ms on paddle hits, and 1500 Hz for 100 ms on scoring events.

7.2 Visual (LEDs)

Two LEDs provide score feedback. The green LED on PD4 flashes 3 times with 100 ms on/off intervals when the player scores, while the red LED on PD7 flashes in the same pattern when the computer scores.

7.3 Haptic (Motor)

A DC motor on PD5 uses Timer0 Fast PWM (OC0B output) for victory celebration. The motor ramps from 0 to full speed over approximately 1 second, holds for 1 second, then ramps back down:

```c
void spin_motor(void) {
    for (uint8_t s = 0; s < 255; s += 5) {
        OCR0B = s;  _delay_ms(20);
    }
    OCR0B = 255;  _delay_ms(1000);
    for (int16_t s = 255; s >= 0; s -= 5) {
        OCR0B = s;  _delay_ms(20);
    }
    OCR0B = 0;
}
```

Listing 9: Motor victory animation

8 Rendering Pipeline

The main loop runs at approximately 50 FPS with a 20 ms frame period. Each frame begins by erasing the old paddle position, drawing a black block over the previous coordinates. The player paddle is then updated from either wireless or joystick input, followed by the computer paddle's AI update (which fires every 10th tick). The ball position is updated next—its old circle is erased and the velocity vector is applied. Paddle–ball collisions and scoring conditions are checked, and finally the paddles, ball, and score overlay are redrawn.

This selective erasure approach—only redrawing changed regions rather than clearing the full screen each frame—avoids visible flicker that would otherwise result from the SPI transfer rate.

9 Pin Mapping Summary

Table 4: ATmega328PB Pin Assignments

Pin	**Function**	**Direction**
PB0	LCD D/CX (Data/Command)	Output
PB1	LCD RST (Reset)	Output
PB2	LCD CS (Chip Select)	Output
PB3	LCD MOSI (Serial Data)	Output
PB5	LCD SCK (Serial Clock)	Output
PD6	LCD Backlight (PWM)	Output
PC4	Joystick X (ADC4)	Input
PC5	Joystick Y (ADC5)	Input
PD0	ESP32 Button Left	Input (pull-up)
PD1	ESP32 Button Right	Input (pull-up)
PD3	Buzzer	Output
PD4	Green LED	Output
PD5	Motor (OC0B PWM)	Output
PD7	Red LED	Output

10 Project Structure

Table 5: Project File Organization

File/Directory	**Description**
`main.c`	ATmega328PB game logic, input handling, rendering
`lib/ST7735.c/.h`	SPI LCD driver for ST7735R controller
`lib/LCD_GFX.c/.h`	Graphics library (circle, line, block, text)
`lib/ASCII_LUT.h`	5×8 ASCII bitmap lookup table
`lab4_pong_blynk_skeleton/`	ESP32 Arduino sketch for Blynk wireless control
`lab4_a.sal`	Saleae Logic Analyzer capture file