LED PING PONG GAME

CS132 COURSEWORK 2

SERENA JACOB, TAHA KHAN
Department of Computer Science

1. Introduction & Device Overview (LED Panel Ping Pong)

Our objective was to create the 2 player game Pong using joysticks as input and the LED panel as the output device.

The LED panel is serially driven, as pixel data is sent as a bitstream into shift registers, with a clock signal used to shift bits into the registers sequentially and a latch signal used to transfer the shifted data to the output stage that drives the LEDs. The panel also has row address lines (A–D) which select the currently active row during multiplexed refresh. In our implementation, these control signals are generated directly using the microcontroller's GPIO pins (e.g., `INP_PIN` for data, `CLK_PIN` for clock, `LAT_PIN` for latch, and `A_PIN`–`D_PIN` for row selection).

The project is split into two folders Emulated and Lab:

The **Emulated folder** contains: *game.c, panel.h, panel_emu.c, emulator.js and index.html (and game.js and game.wasm once compiled).* To compile the code in this folder the Emscripten library (converting C code to WebAssembly for it to be run in the browser). Following command to compile (once library set up):

```
emcc game.c panel_emu.c -O2 -sASYNCIFY -sALLOW_MEMORY_GROWTH -o pong.js
```

To run the game: `python -m http.server 8000`

The **Lab Folder** contains: *game.c, panel.h, panel_hw.c, MakeFile, rules.mk, genstm32 (ledpanel.bin, ledpanel.elf once compiled).* To compile use `make` and to run use: `st-flash --reset write ledpanel.bin 0x8000000`

2. Methodology & Reverse Engineering

The process began with analysing the LED panel's structure, identifying its grid-based addressing mechanism and understanding the data flow between the microcontroller and display driver. Through reverse engineering, we mapped how binary values were transmitted to control individual LEDs. We verified pin connections and tested basic lighting sequences before integrating full game logic.

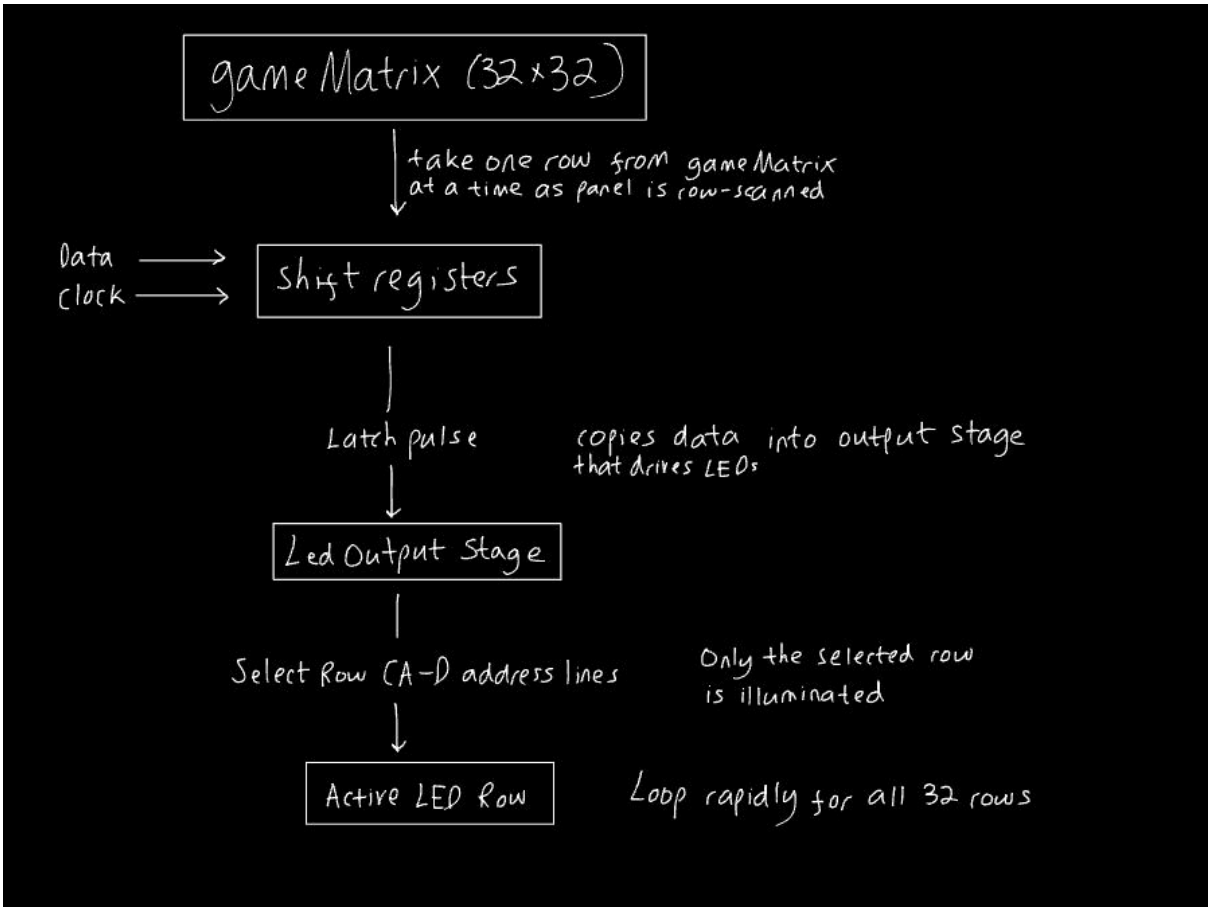
The process began by exploring how the LED panel worked and understanding the data flow between the microcontroller and panel's driver circuitry. We experimented with loops by attempting to update several rows in sequence on both halves of the panel to test the refresh behaviour of the panel. We also varied the RGB bit patterns that were sent to the panel which helped us understand how colour is encoded and that different data sequences create the appropriate matching LED colour.

These experiments allowed us to figure out how to integrate the full game logic and we also designed the visual layout of our game, specifying details about the dimensions of the walls, net, paddles and ball within the 32x32 grid. Our initial hypothesis was that the LED panel used a shift-register interface, where the first bit is put on the data line, and shifted in by the clock and after a row of bits have been shifted the latch is pulsed to transfer the contents to the output and a row is then chosen by setting the A-D address lines. This process is then repeated in quick succession for the remaining rows to refresh the full display on the panel. Instead of relying on full documentation, we read the existing code in the LED example file to help us understand which pins are used, the order that signals are sent in and what the purposes of the various functions were.

From conducting our tests, we found row colour data is sent serially into the shift registers, the latch is pulsed to apply that data to the outputs, and the active row is selected using the A–D address lines.

The diagram below clearly illustrates the row-scanning process used to drive the 32x32 LED panel.

Figure 1: Simplified LED panel row-scanning protocol showing serial data transfer, latching, and row selection. Only one row is active at any time, requiring continuous high-speed refreshing.



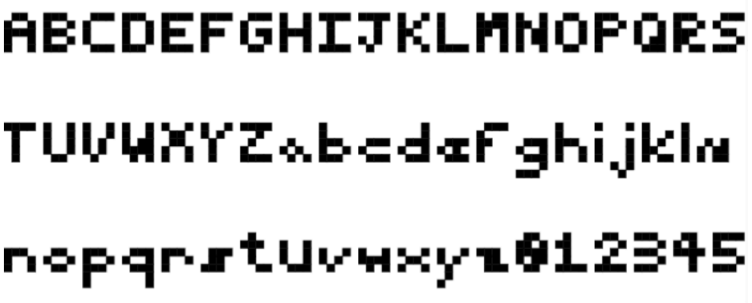
3. Software Design & Abstraction

Our software design aimed to separate low-level LED panel control from high-level game logic, so the game could be developed without constantly dealing with GPIO signalling. This was important because the LED panel is row-scanned, which requires the firmware to repeatedly refresh rows at high frequency to keep the image visible. A further abstraction layer separates the game logic from the panel-specific implementation via the panel.h module, which defines a consistent interface for display and input functions. This interface is implemented by panel_emu.c in the Emulated folder and panel_hw.c in the Lab folder, containing contain functions such as setupPanel(), setupInput(), getRawInput(), delay_ms(), PrepareLatch(), LatchRegister(), SelectRow(), PushBit(), and ClearRow()

In the 32×32 frame buffer, each entry stores a colour code. ("X", "W", "R", "G", "B", "C", "Y", "M"). These are mapped to RGB on/off patterns using the colours [ ] [ ] lookup table `e.g. R = [1, 0, 0], B = [0, 1, 0], C = [0, 1, 1].` (Not explicitly defined in this way but rather given their own index in a 2D array).

Numbers and letters were encoded into 3 dimensional arrays with each individual character represented by a 2 dimensional array. This can be thought of as a matrix of 0s and 1s

representing whether that pixel should be on or off. The following fonts were used as inspiration for the characters. (1)



This makes the game logic simple, with drawing functions updating the grid rather than directly controlling pins. During output, these codes are converted into RGB bit patterns.

The encoding for the characters interacts well with the underlying 32x32 framebuffer. The gameMatrix object can be thought of as a grid of pixels (the y axis going in the opposite direction). This allows for easy integration with the physics of the game.

Functions are organised into layers:

- **Rendering layer** (drawBall(), drawPaddles(), drawBorders(), drawWinBorders() drawNet(), displayScores(), displayStart(), displayWinner()) -> writes only to gameMatrix.
- **Game update layer** (updateBall(), detectCollisions(), detectPointWin(), updatePaddlePositions()) -> updates the state (positions/scores) and triggers redraws.
- **Driver layer** (updateDisplay(), displayRow(), PushBit(), SelectRow(), PrepareLatch(), LatchRegister()) -> implements the row-scanning routine.

This abstraction allows the main program to draw in a high-level manner by updating the frame buffer, then calling updateDisplay() to show it on the panel, without needing to manage clock/latch timing or the A–D row address lines, as all of this is handled inside the driver layer.

The modular abstraction not only allowed the game logic to be tested in the emulator without needing the physical hardware to test the functions that interface with it, but also makes the panel interface functions reusable for other projects.

Main functions

- **main()** – Initialises the panel and ADC (setupPanel(), setupInput()), then switches to startScreen(), mainGame()or winScreen() based on gameMode.

- **startScreen()** – Clears the start layout (initGameMatrix(), drawBorders(), displayStart()) and keeps it visible using updateDisplay().
- **mainGame()** – Runs the gameplay loop, updating paddles (updatePaddlePositions()), moving the ball (updateBall()), handling collisions (detectCollisions()), updating the score (detectPointWin()), redrawing the frame buffer, then refreshing the panel (updateDisplay()).
- **winScreen()** – Displays the winning message (handleWin()/displayWinner()), refreshes the panel, then resets the state for a new game.

Hardware initialisation

- **setupPanel()** – Configures GPIO output pins for panel control.
- **setupInput()** – Configures ADC for joystick sampling
- **delay_ms(uint32_t ms)** – Pauses the microcontroller by looping, controlling the refresh rate and timing between clock and latch pulses.

Display refresh functions

- **updateDisplay()** – Implements row-scanning refresh. For each row address, shifts row data, latches it, and repeats rapidly so the full image appears.
- **displayRow(char matrixRow[ ])** – Converts a 32-pixel row from gameMatrix into RGB bit values (using colours [ ] [ ]) and shifts the data out to the panel.
- **PushBit(int onoff)** – Sets the data line and then pulses the clock to shift a single bit into the panel.
- **PrepareLatch() / LatchRegister()** – Controls when shifted data is applied to the visible output.
- **SelectRow(int row)** – Sets A–D address lines to select the active row.
- **ClearRow(int row)** – Clears a row by shifting zeros to reduce unwanted visual effects (artefacts) before writing new data.

Frame buffer drawing (visuals)

- **initGameMatrix()** – Clears the 32×32 frame buffer to 'X' (off).
- **drawBorders() / drawNet()** – Draws the static playfield elements, including the walls and centre line.
- **drawBall() / eraseOldBall()** – Draws the ball and clears its previous position.
- **drawPaddles() / drawPaddle() / eraseOldPaddles()** – Updates paddle pixels based on new/old positions.
- **displayScores() / drawDigit()** – Displays the score using predefined digit bitmaps.
- **displayStart() / displayWinner() / drawCharacter() / handleWin()** – Displays text screens using predefined character bitmaps.
- **tempDisplay()** – Prints gameMatrix to the console for debugging when the physical display output is unreliable.

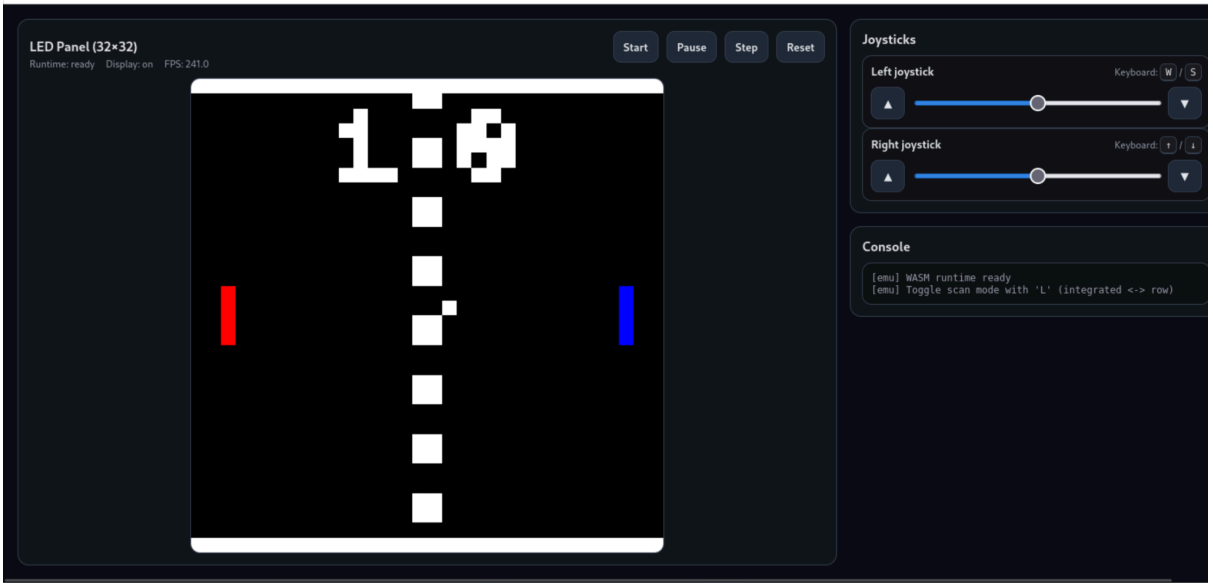
Game logic

- **initGame()** – Sets initial positions for the paddles and ball, as well as their velocities and the serve direction.
- **updateBall()** – Updates the ball's position (ballX, ballY) based on its velocity.
- **detectCollisions()** – Handles ball collisions with paddles or borders, adjusting its direction
- **detectPointWin()** – Detects when the ball crosses a scoring boundary and updates the score and game state.

ADC/joystick input

- **getRawInput(int channelValue)** – Reads a single ADC value from the requested channel.
- **getRawPaddleInput(int whichPaddle)** – Reads the joystick channels for the selected paddle and returns the resulting input value.
- **convertInputToPaddlePosition(int inputValue)** – Converts ADC readings into valid, scaled paddle Y-positions
- **updatePaddlePositions()** – Updates both paddles by reading ADC values and converting to positions.
- **inputCheck(...)** – Normalises joystick input using threshold checks to distinguish between neutral and movement states.

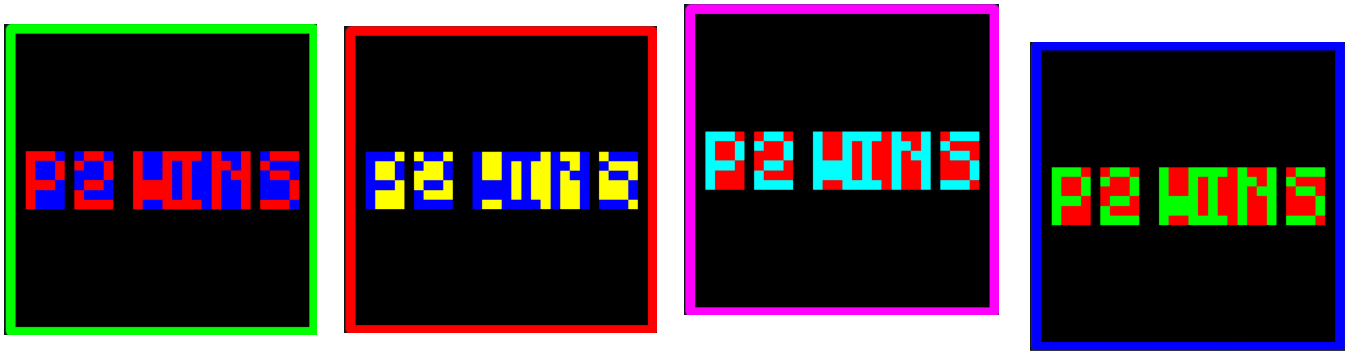
Figure 3: As an extension of this modular design, we also developed a web-based simulator for the LED panel.



This emulator was also used to display what the game should look like if framebuffer is outputted perfectly onto the LED panel (assuming no errors in game logic or panel_hw.c)

Start screen:

Win Screen alternating colours rapidly:



4. Reflection on Process and Outcomes

The project achieved most of the intended functionality, but the LED output was not fully stable by the end. Although the frame buffer contained the correct image, only the final row appeared on the panel. We identified this as a row-scanning issue: only one row can be active at a time, so the firmware must continuously loop through all rows at a high refresh rate for the full image to appear stable.

When hardware was no longer accessible, we decided to develop an emulator to continue testing. The emulator is a single-page web app (index.html, emulator.js) with panel_emu.c, providing the same interface used by the game. It has two modes (toggled with ‘L’): ‘interface’ mode, which displays the frame buffer directly and ‘row’ mode, which mimics row-scanning by showing only the currently selected row. Although the emulator was effective for validating game logic and visuals it could not fully replicate the physical panel’s behaviour in row mode, meaning that timing-dependent parameters like refreshDelay

(between rows) still require hardware testing. This limitation arises because the real LED panel exhibits persistence between updates, with pixels fading rather than instantly switching off, whereas the emulator's pixels switch off immediately. A clear improvement would be to model this persistence by adding a fading effect between row updates. More hardware time would have been needed to compare how different refresh delays affect the overall display stability.

5. Conclusion and Future Work

This project produced a structured software solution for a Ping Pong game on a 32×32 LED matrix. A key outcome was building a 32×32 frame buffer model for the display, which allowed us to implement and test the core gameplay, including joystick-controlled paddles and ball movement, without constantly dealing with low-level GPIO details.

The flagship product from this coursework was the emulator. It provided an innovative solution limited hardware access and enabled detailed debugging and testing of game functionality. Future improvements would focus on implementing a fade effect, so the emulator better matches the physical panel's behaviour and allows testing of timing parameters such as the inter-row refresh delay. This would not only allow for testing of game logic but also of how that logic then interacts with the display under realistic refresh conditions. A further extension would be to create a custom library that maps the stm32 functions onto the emulator directly instead of needing a separate interface. This would allow for deeper debugging without maintaining separate implementations.

For the LED panel itself, future work would first focus on producing a stable full-screen display by optimising and validating the refresh routine. Once this is reliable, we would introduce smoother animation and adjustable difficulty, such as by increasing the speed of the ball to make the gameplay more challenging. Overall, this project delivered a strong foundation and completing the refresh routine would make it a polished and fully playable LED-matrix game.

6. References

1. https://community.arduboy.com/t/4x6-font-for-the-arduboy/5589
2. https://fontstruct.com/fontstructions/show/505134/4x5_pixel_font