Sensors and Microsystems Electronics

Endless-Runner — Project for uC labs

Student: *GURLUI Octavian*

April 21, 2025

1 Introduction

The aim of the project was to create a fully interrupt-driven endless runner game on the provided development board, by making use of the display consisting of a matrix of 80x7 LEDs and the joystick as an input. As a high level overview, it has been intended to have a 'character' show up on screen (represented by a specific LED pattern). By pressing the joystick, the character would move up and down on the screen. A set of randomly-generated obstacles (represented by specific LED patterns) would move towards the character at a designated pace. An obstacle which made it past the character without collision would result in a score increment, whereas an obstacle which collides with the character would end up in the game being over, the score being shown on screen, and an option to restart the game by pressing any key.

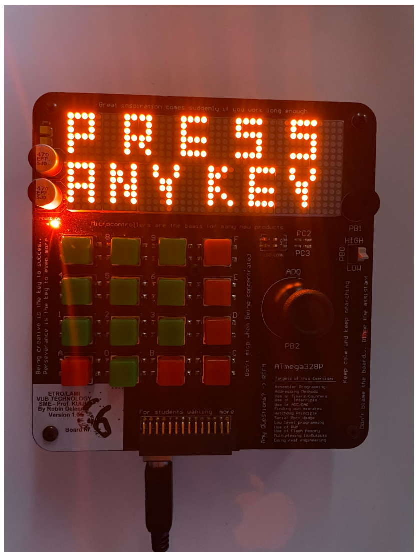

Figure 1: Game start splash screen

The initial requirements included a display refresh timer, distinct from a game logic update timer, keyboard debouncing, robust SRAM management, and proper usage of the screen buffer.

Although some features such as EEPROM-backed scores and font rendering have been postponed due to time constraints, the core objectives of smooth animation, responsive controls and reliable collision detection were fully achieved.

2 Keyboard logic

Instead of having the logic to read the joystick inputs (which required using the ADC and using some thresholds to distinguish between up/neutral/down position), an implementation making use of the keyboard was first considered. Due to timing constraints, the joystick has finally not been used.

Without going into depth regarding this part, since it has been covered during the labs, it is simply noted that the "two-step" scan method was implemented, in a self contained function.

To avoid phantom presses, some short NOP delays were added after each PORT to DDR transition to let I/O buffers settle and avoid metastability. Finally, the entire keypad function was wrapped in a function that PUSHes every modified register onto stack and then POPs them before returning, except for the two registers which are used in the main loop to determine which key was pressed. Using the stack was necessary to avoid the display function interfere with the keyboard logic and viceversa, since many of the registers R16-R31 are re-used in both functions.

3 Screen logic

Driving the 80x7 LED matrix requires making use of the shift registers and a dedicated DisplayPattern subroutine which is invoked each millisecond by timer0's overflow. The implementation relies on the "screen buffer" method. Therefore, a 70-byte space in the SRAM memory is allocated as the screenbuffer, and pointer arithmetic is used to read from the screenbuffer.

The initial implementation produced scrambled patterns because the Z pointer was traversing the screen buffer in the wrong direction, and the bit-mask rotation logic I was using was flawed. After switching to a post-increment pointer arithmetic and replacing the mask loop with a simple LSR on a fresh mask register for each bit, the columns lit up in the correct sequence.

Another issue which needed attention was the fact that the display routine could be invoked when the 'game logic' was building the screen buffer, and even with registers being pushed and popped on the stack, this lead to registers being corrupted. This was fixed by wrapping the DisplayPattern call and the function which copies the Obstacle Map and Character Map into screen buffer (which will be discussed later) with CLI/SEI to not allow any timer overflow interrupts to happen until the buffer writes were complete.

4 Character Movement

Vertical movement of the 4x4 pixel character presented a mild challenge because of the earlier-made design choice of writing the displayPatter function without accounting for the LED matrix's memory layout which splits at midpoint (a “row” of 80 columns in memory represents, in fact, two physical rows of 40 LEDs). To move the character smoothly and clamp it within the visible area, four byte-offset variables have been stored in SRAM (characterOffset1...4), each pointing to one row of the 4-pixel tall character.

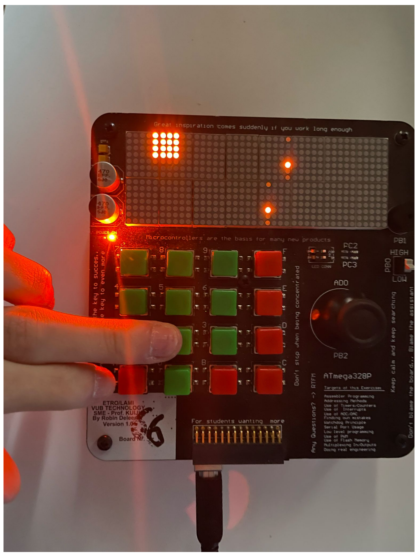
Figure 2: Character Up upon pressing up key.

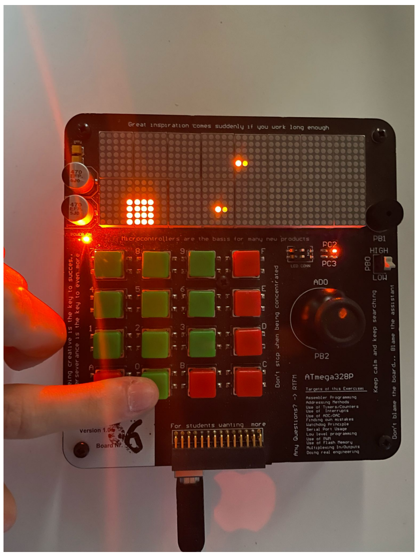
Figure 3: Character Down upon pressing down key.

On an Up or Down key press (which can be seen in fig.2 and fig.2, the characterMovesUp/characterMovesDown routine performs the following steps in register-safe subroutines: first clears out the old characterMap by writing zeros to the four addresses using pointer arithmetic, then it computes each new offset for the 4 rows by adding/subtracting 10, while also accounting for the “split” at the midpoint of the display and ensuring the character stays within the boundaries of the display. For example: if the initial position of the character is defined by offsets 50,60,5,15 (meaning the character is “in the middle” of the display vertically, and at the first 8 bits horizontally), upon pressing the down key, the character should be defined by a set of offsets (60,5,15,25). The routine reloads the bit pattern corresponding to the ‘physical shape’ of our character, a square, and updates the character map accordingly, and this is the only way the character map gets updated within the code - by a press of the designated Up and Down keys.

5 Obstacle Generation

Obstacles are managed in parallel via a separate 70-byte buffer, obstacleMap, and 4 independent sets of state variables in the SRAM. One single-bit pattern register (a single bit = a single LED will be lit on, corresponding to an obstacle), a horizontal offset (representing which line the obstacle will be placed on), and a countdown delay for each obstacle.

On each game-logic tick (timer1 overflow, which happens at a slower frequency than the screenBuffer update - allowing the user to actually see the obstacle moving across the screen i.e. the obstacles will not 'teleport' from left to right too fast for the user to be able to react), the UpdateObstacleMap routine iterates through all four obstacles. If an obstacle's delay is still positive, it decrements the delay. Once the delay reaches zero, the obstacles get spawned at some locations which are indirectly determined by the game logic timer (essentially the obstacles always perform the same movement across the map, but only become visible to user, and only appear in the obstacle map once the 'delay' associated to them reaches zero).

The 'movement' of the obstacles towards the character is done by shifting the pattern byte left (done with LSR due to how the bits are actually fed into the display buffer) to advance the lit LED within 8 bits, followed by reinitializing the pattern one "offset" to the left and shifting the pattern bit by bit again. Essentially, when the bit pattern underflows to zero, the code reloads it to 0b10000000 and decrements the offset by one, wrapping from column 0 of previous offset, to column 8 of current offset. Like the character routines, obstacle updates push and pop all modified registers so they can safely run at any point in the main loop without corrupting the display driver.

6 Collision Detection

This feature can be found directly within WriteNewPatternToBuffer, since this is the place where the merge between characterMap and obstacleMap occurs such that both character and obstacle(s) are copied into screenBuffer. As both maps are traversed using pointer arithmetic, the algorithm checks for an "OR" between the pairs of bytes to build the frame fed into screenBuffer, but also checks for an "AND" between the pairs of bytes to see if any obstacles overlap with the character. If the result of the latter is nonzero, the routine immediatly sets a global gameOverFlag, and on the next frame we jump to a routine that handles the game over scenario. This approach to colission detection is straightforward and avoids any timing mismatches, as the same interrupt-driven update that delivers the new frame to LEDs also enforces the collision rules at the same time instance.

7 Known bugs, potential fixes, further potential improvements

- Score keeping and difficulty levels: While the infrastructure for a score counter exists, it never made it into EEPROM or onto the display. Writing the score into EEPROM on game over, and allowing difficulty selection at the splash screen to adjust both obstacle and movement timers, would round out the user experience.

- The current "random" delays are hardcoded counters and can create impossible obstacle patterns. A true pseudo-random generator plus some simple spacing rules (minimum delay between spawns) would produce varied but fair obstacle sequences.

- Simulatenous Up/Down key presses can corrupt the state of the character.