The snake game on the Atmega32U4

Jayendra Ellamathy

8th March 2021

Contents

1 Introduction

The kids who grew up in the 90's fondly remember spending countless hours playing the snake game on the first mobile handsets. Nowadays, phones and the games have evolved to a complex stage but still a part of us wants to return to the nostalgia you get while playing this simple game. Here, we implement the snake game on a 8×8 LED dot matrix with the Atmegs32U4 micro-controller. The direction of gameplay is controlled using push buttons representing the 4 directions - UP, DOWN, LEFT, RIGHT. The implementation itself is under 500 lines of code. So, connect, flash, boot, and take a journey back in time!

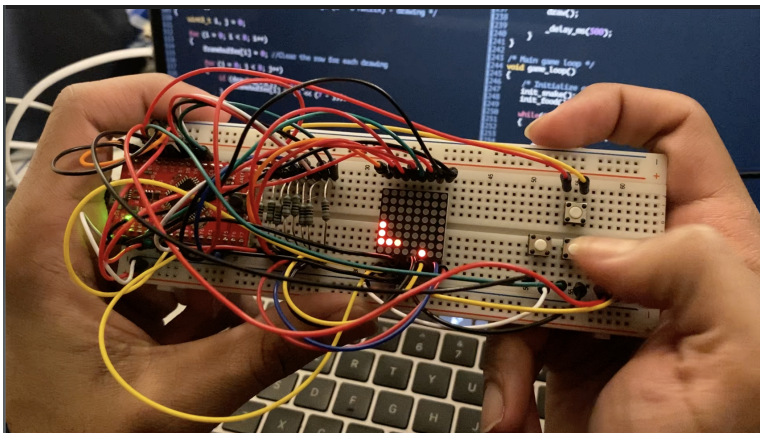


Figure 1: Connect, flash, boot, and take a journey back in time!

2 Abstract

The report contains the implementation of the popular snake game on a 8×8 dot matrix LED with the Atmega32U4 MCU. The report explains the hardware implementation - Interfacing the push buttons for controlling the direction of gameplay, interfacing the LED dot matrix to display the game. Then the report explains the software implementation of both the snake game and the MCU control code - Interrupts, timers, etc. The design choices taken for the implementation are explained along the way in both parts.

3 Hardware

3.1 Pin diagram

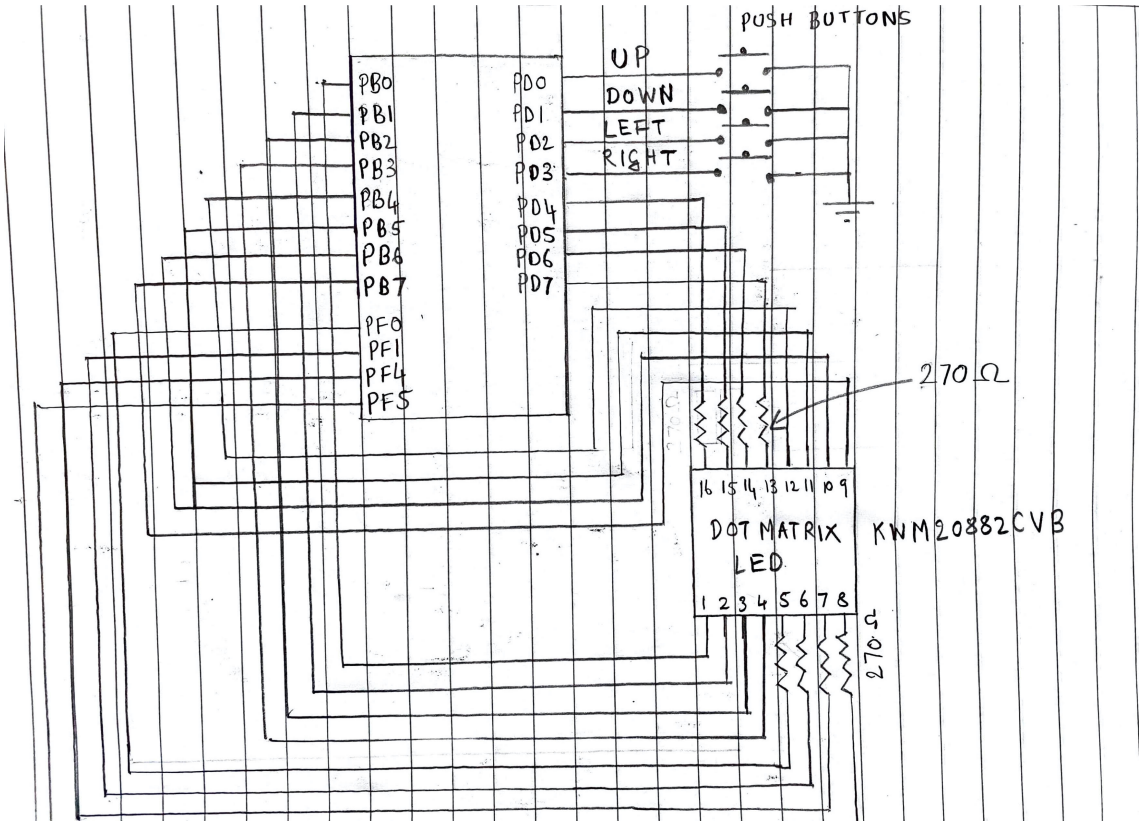


Figure 2: Connect, flash, boot, and take a journey back in time!

3.2 Push buttons

The direction of movement of the snake is controlled by 4 push buttons. The push buttons are intuitively placed to represent the 4 possible directions - UP, DOWN, LEFT, RIGHT.

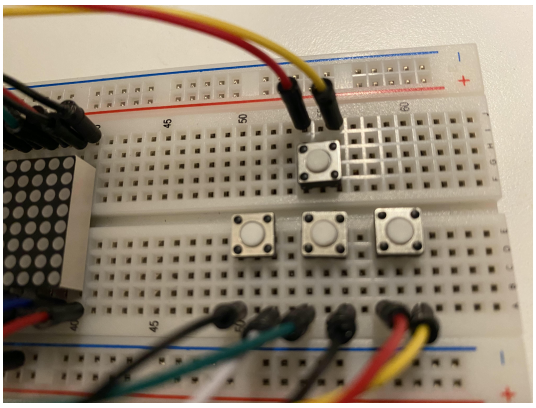


Figure 3: Use of push buttons to control direction of gameplay

The Atmega32U4 accommodates 4 external hardware interrupts on pins PD0, PD1, PD2, and PD3 which are referenced as INT0, INT1, INT2, and INT3 respectively.

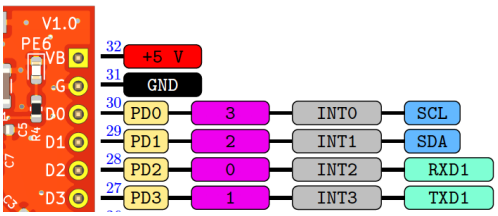


Figure 4: External interrupts on the Edduino

Whenever a change is detected on these pins, an interrupt is raised and the MCU control jumps to the interrupt service routine(ISR). The current context of execution of the program is saved, the ISR is executed and then the control is transferred back to the MCU program.

This facility perfectly serves the use case of the push button. Whenever a push button is pressed, an ISR is invoked which changes the direction of the snake. A button de-bounce delay can be added to the ISR to handle unnecessary inputs from the button.

3.3 Dot matrix LED display

Here, we have used a 8×8 LED dot matrix, Part Number: KWM20882CVB.

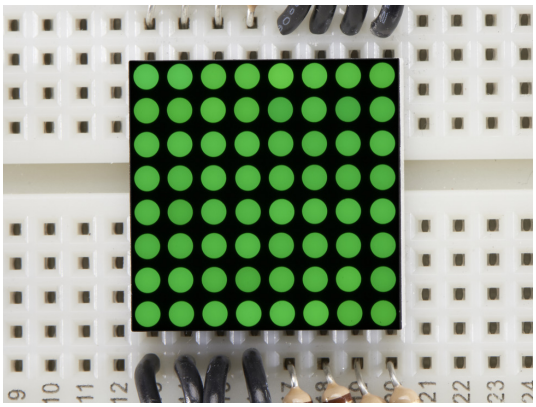


Figure 5: 8×8 LED dot matrix

The pin configuration of matrix is shown in the figures below. Each dot on the matrix is switched ON/OFF using two GPIO pins of the micro-controller.

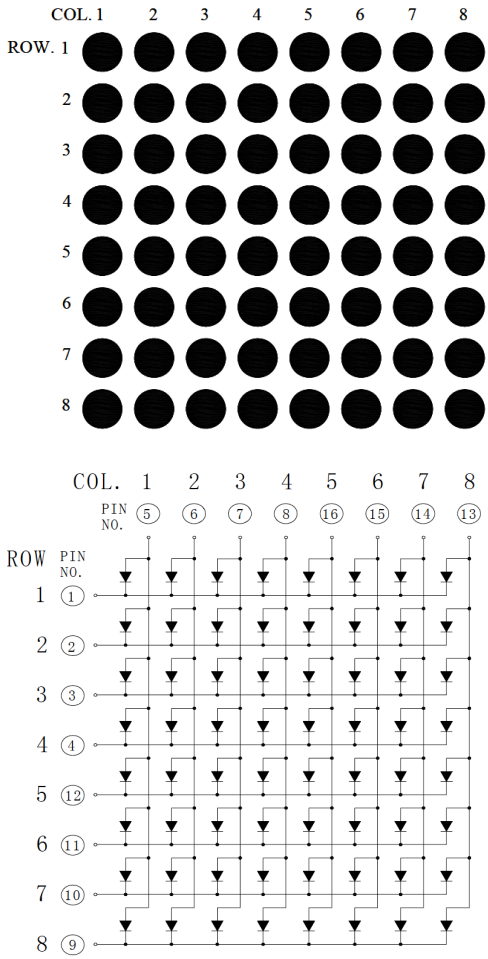


Figure 6: Pin configuration of the LED matrix

With 16 pins of the micro-controller we can control every LED of the matrix by pulling the pins up or down through the software.

4 software

4.1 The snake game

4.1.1 data-structures

- The snake is represented as a linked list with node structure as:

```
typedef struct snake {
```

```
    struct position pos;
    struct snake *next;
    struct snake *prev;
} snake_t;
```

- Each position is represented as x and y co-ordinates in 2 dimensional array of size 8×8.

```
typedef struct position {
    uint8_t x;
    uint8_t y;
} position_t;
```

- The entire game is represented on an 2 dimensional array of size 8×8 as this mimics the actual LED dot matrix.

```
uint8_t drawing[8][8];
```

- Food for the snake takes one dot on the matrix and hence is represented as an x and y co-ordinate.

```
position_t food;
```

- The current game direction is stored in a global variable. This global variable will be changed by the ISR tied to the push buttons:

```
typedef enum dir {
    UP,
    DOWN,
    LEFT,
    RIGHT,
} dir_t;

dir_t direction;
```

4.1.2 Functions

- `void init_snake()`: Creates the snake at the initial position at the start of the game with body length of 2.
- `void add_to_snake(uint8_t x, uint8_t y)`: Add a snake body unit at the head position of the snake. Update the snake when food is eaten.
- `void draw_snake()`: Draw the snake from the linked list on to the 8 * 8 matrix array - drawing.
- `void destroy_snake()`: Free up memory allocated for the snake.
- `void init_food()`: Initialize position of food at (4, 4).

- `void generate_food()`: Generate food at random position. Ensures food does not collide with snake.
- `void draw_food()`: Draw the food onto the 8 * 8 matrix array - Drawing.
- `void calculate_next_move(uint8_t *x, uint8_t *y)`: Calculate the next move of the snake based on the direction.
- `int check_valid_move(uint8_t x, uint8_t y)`: Check if the x and y co-ordinates collide with that of the snake body.
- `int update_game()`: Check current state and create the next state of the game.
- `void blink_game_over()`: Blink the drawing to indicate the game is over.
- `void game_loop()`: Main game loop.

4.2 Controlling the buttons

The buttons are tied to the external hardware interrupts INT0..3 of the microcontroller through pins PD0..3. The ISR of each button changes the global value `direction` which changes the direction of the snake in the next game state.

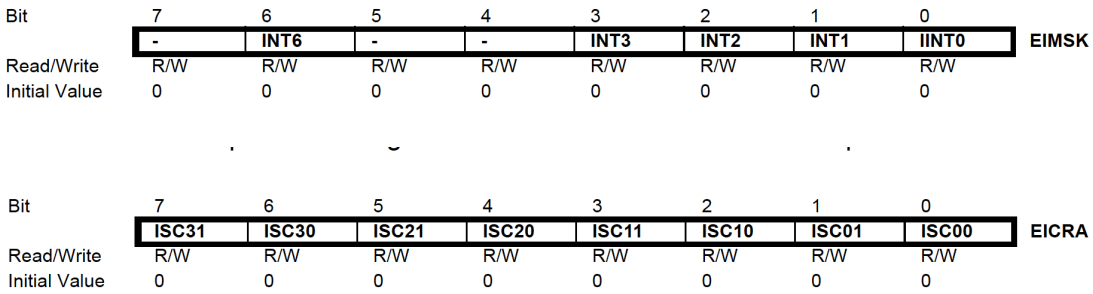
Eg: ISR code of button connected to PD0:

```
ISR(INT0_vect)
{
    direction = DOWN;
}
```

For this, first we declare PD0, PD1, PD2, PD3 as input GPIO pins. Then we also set them all to HIGH(the other end of the button will be connected to GND).

```
DDRD = 0b11110000;

PORTD = 0b00001111; //Pull-up to HIGH
```



Bit	7	6	5	4	3	2	1	0	
	-	INT6	-	-	INT3	INT2	INT1	IINT0	EIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 7: Register to control external HW interrupts

ISC01	ISC00		Description
0	0		The low level on the INT0 pin generates an interrupt request.
0	1		Any logical change on the INT0 pin generates an interrupt request.
1	0		The falling edge on the INT0 pin generates an interrupt request.
1	1		The rising edge on the INT0 pin generates an interrupt request.

Figure 8: ISC bits define the level or edge that triggers the INT0

Then we enable interrupts INT0..3. We also trigger an interrupt on the rising edge by controlling the pin sense bits ISCXX.

```
/* Set interrupt mask for buttons */
EIMSK |= (1<<INT0) | (1<<INT1) | (1<<INT2) | (1<<INT3);
EICRA = (1<<ISC01 | 1<<ISC00) | (1<<ISC11
| 1<<ISC10) | (1<<ISC21 | 1<<ISC20) | (1<<ISC31 | 1<<ISC30);
/* Trigger on rising edge */
```

4.3 Drawing on the LED matrix

Now that we have our drawing on the 2 dimensional array of size 8 we translate the same into our 8×8 dot matrix.

With the current pin configurations we can only draw one row at a time as we have to selectively switch ON/OFF our LEDs. However, if we do this quick enough we can render the entire image as our human eye is not quick enough to differentiate the MCU rendering the image row by row.

Here, we will use the timer overflow interrupt to render row by row. Every time the interrupt occurs, we change the output signal on the ports to light up the next row. The bit pattern of each element of `framebuffer` represents each row of the LED matrix.

```
volatile uint8_t framebuffer[8];

/* Interrupt service routine to handle one row at a time.
 * Within 8 interrupts, all 8 rows of the LED matrix will be covered. */
    ISR(TIMER0_OVF_vect)
    //Timer based interrupt - occurs every 0 to 255 count of timer
    {
        static uint8_t digit;

        PORTB = 0b11111111; //Clear all Rows
        PORTB = ~(0b00000001 << digit);  //Set current Row

        //Set columns
        PORTD = (framebuffer[digit] & 0b11110000) | 0b00001111;
        PORTF = (framebuffer[digit] & 0b00000011)
        | ((framebuffer[digit] & 0b00001100) << 2);

        digit++;
        digit = digit % 8; //Repeat from start
    }
```

We also need to set the appropriate bits to configure our times with the clock speed that we need. Additionally, we need to enable the timer interrupt.

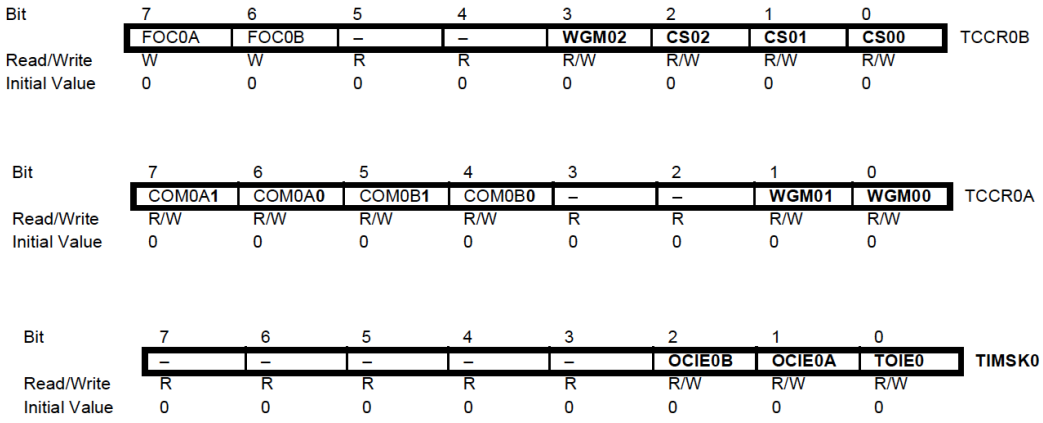


Figure 9: Registers for setting timer properties and timer interrupt

```
TCCR0A = (0 << COM0A1) | (0 << COM0A0) | (0 << COM0B1) | (0 << COM0B0)
    | (0 << WGM01)  | (0 << WGM00);
TCCR0B = (0 << WGM02)
    | (0 << CS02)   | (1 << CS01)   | (1 << CS00); //Clock prescaler
TIMSK0 = (0 << OCIE0B) | (0 << OCIE0A) | (1 << TOIE0);
```

4.4 main

The main loop consists of checking the current state of the game and updating to the next state and then drawing the image onto the 2 dimensional array. We check the current snake and food position and check if the next is a valid move or not. We generate the food at a random position if the snake has consumed in the previous state.

5 source code

The code is split into 2 logical parts.

- snake.c and snake.h contain the logic for the game.
- main.c contains everything else - From Timers, interrupts, etc.

5.1 snake.h

```
#include <stdlib.h>
#include <avr/io.h>
```

```c
#include <util/delay.h>


/* x and y co-ordinates for 8 * 8 matrix image */
typedef struct position {
    uint8_t x;
    uint8_t y;
} position_t;

/* Linked list struct to represent snake */
typedef struct snake {
    struct position pos;
    struct snake *next;
    struct snake *prev;
} snake_t;

/* Game directions */
typedef enum dir {
    UP,
    DOWN,
    LEFT,
    RIGHT,
} dir_t;

extern void draw();


/* Creates the snake at the initial position at the start of the game */
void init_snake();

/* Add a snake body unit at the head position of the snake. Update snake when
 * food is eaten */
void add_to_snake(uint8_t x, uint8_t y);

/* Draw the snake from the linked list on to the 8 * 8 matrix array
- drawing */
void draw_snake();

/* Free up memory allocated for the snake */
void destroy_snake();

/* Init position of food at (4, 4)*/
void init_food();

/* generate food at random position.
Ensures food does not collide with snake. */
```

```
void generate_food();

/* Draw the food onto the 8 * 8 matrix array */
void draw_food();

/* Calculate the next move of the snake based on the direction */
void calculate_next_move(uint8_t *x, uint8_t *y);

/* Check if the x and y co-ordinates collide with
that of the snake body */
int check_valid_move(uint8_t x, uint8_t y);

/* Check current state and create the next state of the game */
int update_game();

/* Blink the drawing to indicate the game is over */
void blink_game_over();

/* Main game loop */
void game_loop();
```

5.2 snake.c

```
#include "snake.h"


extern uint8_t drawing[8][8];

/* Snake is a linked list starting with HEAD snake_head */
snake_t *snake_head = NULL;
snake_t *snake_tail = NULL;

/* Current position of the food */
position_t food;

/* Current direction of the game */
dir_t direction;

/* Creates the snake at the initial position at the start of the game */
void init_snake()
{
    /* Initialize snake with head + 1 body unit */
    snake_head = (snake_t *) malloc(sizeof(snake_t));
    snake_tail = (snake_t *) malloc(sizeof(snake_t));
```

```
    /* Start snake game with snake at (2, 2) and (2, 3) */
    snake_head->pos.x = 2;
    snake_head->pos.y = 2;
    snake_head->next = snake_tail;
    snake_head->prev = NULL;

    snake_tail->pos.x = 3;
    snake_tail->pos.y = 2;
    snake_tail->next = NULL;
    snake_tail->prev = snake_head;
}

/* Add a snake body unit at the head position of the snake. Update snake when
 * food is eaten */
void add_to_snake(uint8_t x, uint8_t y)
{
    snake_t *snake;

    snake = (snake_t *) malloc(sizeof(snake_t));

    snake_head->prev = snake;
    snake->next = snake_head;
    snake->prev = NULL;
    snake_head = snake;
}

/* Draw the snake from the linked list on to the 8 * 8 matrix array
- drawing */
void draw_snake()
{
    snake_t *snake;

    snake = snake_head;

    while (snake != NULL) {
        drawing[snake->pos.x][snake->pos.y] = 1;
        snake = snake->next;
    }
}

/* Free up memory allocated for the snake */
void destroy_snake()
{
    snake_t *snake;
    snake_t *next;
```

```
    snake = snake_head;
    next = snake->next;

    while (snake != NULL) {
        free(snake);
        snake = next;
        next = snake->next;
    }

    snake_head = NULL;
    snake_tail = NULL;
}

/* Start game with food at the (4, 4) */
void init_food()
{
    food.x = 4;
    food.y = 4;
}

/* generate food at random position.
Ensures food does not collide with snake. */
void generate_food()
{
    uint8_t x, y = 0;

    while (1) {
        x = rand() % 8;
        y = rand() % 8;

        if (drawing[x][y] == 1) {
            //Snake body - Hence generate at another location
        } else {
            food.x = x;
            food.y = y;
            break;
        }
    }
}

/* Draw the food onto the 8 * 8 matrix array - Drawing */
void draw_food()
{
    drawing[food.x][food.y] = 1;
}
```

```
/* Calculate the next move of the snake based on the direction */
void calculate_next_move(uint8_t *x, uint8_t *y)
{
    //Calculate movement by 1 position
    switch (direction)
    {
        case UP:
            if ((*y - 1) < 0) {
            //If you go over edges then round up the value
                *y = 7;
            } else {
                *y = *y - 1;
            }
            break;
        case DOWN:
            if ((*y + 1) > 7) {
                *y = 0;
            } else {
                *y = *y + 1;
            }
            break;
        case LEFT:
            if ((*x - 1) < 0) {
                *x = 7;
            } else {
                *x = *x - 1;
            }
            break;
        case RIGHT:
            if ((*x + 1) > 7) {
                *x = 0;
            } else {
                *x = *x + 1;
            }
            break;
        default:
            return;
    }
}

/* Check if the x and y co-ordinates collide with that of the snake body */
int check_valid_move(uint8_t x, uint8_t y)
{
    snake_t *snake;

    snake = snake_head;
```

```
    while (snake != NULL) {
        if ((snake->pos.x == x) && (snake->pos.y == y)) {
            return -1;
        }
        snake = snake->next;
    }

    return 0;
}

/* Update game: Calculate the next state of the game
- Next snake position, food position, game over ?
 * RETURN: -1 if next move ends game, 0 for a valid move */
int update_game()
{
    snake_t *snake;
    uint8_t x,y = 0;


    //Get the current value of head
    x = snake_head->pos.x;
    y = snake_head->pos.y;

    snake = snake_tail;

    //Kick out the last value
    while (snake->prev != NULL)
    {
        snake->pos.x = snake->prev->pos.x;
        snake->pos.y = snake->prev->pos.y;

        snake = snake->prev;
    }

    //Calculate next move based on the direction
    calculate_next_move(&x, &y);

    if (-1 == check_valid_move(x, y)) {
        return -1;
    }

    //Check if next position is food, append the snake if so
    if ((food.x == x) && (food.y == y))
    {
        add_to_snake(x, y);
```

```
            generate_food(); //Since food is consumed,
                             //now add food in new location
        }

        //Update the position of the head
        snake_head->pos.x = x;
        snake_head->pos.y = y;

        return 0;
}

/* Blink the drawing to indicate the game is over */
void blink_game_over()
{
        uint8_t i = 0;

        for (i = 0; i < 6; i++)
        {
            refresh_drawing();

            if (i % 2 == 0) {
                draw_snake();
                draw_food();
            }

            draw();

            _delay_ms(500);
        }
}

/* Main game loop */
void game_loop()
{
        /* Initialize game */
        init_snake();
        init_food();

        while(1)
        {
            refresh_drawing();

            draw_snake();
            draw_food();

            draw();
```

```
        if (-1 == update_game()) {
            blink_game_over();
            destroy_snake();
            return;
        }

        _delay_ms(200);
    }
}
```

5.3 main.c

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include "snake.h"


extern dir_t direction;

/* Array of PORTD output values. Each value drives 1 row of the LED matrix */
volatile uint8_t framebuffer[8];

/* Current image we display on the LED matrix */
uint8_t drawing[8][8] = {0};


/* Interrupt service routine to handle one row at a time.
 * Within 8 interrupts, all 8 rows of the LED matrix will be covered. */
ISR(TIMER0_OVF_vect) //Timer based interrupt - occurs every 0 to 255
count of timer
{
    static uint8_t digit;

    PORTB = 0b11111111; //Clear all Rows
    PORTB = ~(0b00000001 << digit);  //Set current Row

    //Set columns
    PORTD = (framebuffer[digit] & 0b11110000) | 0b00001111;
    PORTF = (framebuffer[digit] & 0b00000011) \
    | ((framebuffer[digit] & 0b00001100) << 2);
```

```
    digit++;
    digit = digit % 8; //Repeat from start
}


/* Button interrupts:
 * Each button is tied up with an external interrupt.
 *
 * INT0: PD0: DOWN
 * INT1: PD1: UP
 * INT2: PD2: LEFT
 * INT3: PD3; RIGHT */
ISR(INT0_vect)
{
    direction = DOWN;
}

ISR(INT1_vect)
{
    direction = UP;
}

ISR(INT2_vect)
{
    direction = LEFT;
}

ISR(INT3_vect)
{
    direction = RIGHT;
}

void init(void)
{
    //Declare pins as output pins for LED and input for buttons
    DDRD = 0b11110000;
    DDRB = 0b11111111;
    DDRF = 0b00110011;

    PORTD = 0b00001111; //Pull-up to HIGH

    /* Set interrupt mask for buttons */
    EIMSK |= (1<<INT0) | (1<<INT1) | (1<<INT2) | (1<<INT3);
    EICRA = (1<<ISC01 | 1<<ISC00) | (1<<ISC11 \
    | 1<<ISC10) | (1<<ISC21 | 1<<ISC20) | (1<<ISC31 | 1<<ISC30);
```

```
/* Trigger on rising edge */

    // we start with
    // normal operation
    // clk/1024 prescaler (about 8 kHz)
    // interrupt on Timer 0 overflow

    TCCR0A = (0 << COM0A1) | (0 << COM0A0) | (0 << COM0B1) | (0 << COM0B0)
    | (0 << WGM01)  | (0 << WGM00);
    TCCR0B = (0 << WGM02)
    | (0 << CS02)   | (1 << CS01)   | (1 << CS00); //Clock prescaler
    TIMSK0 = (0 << OCIE0B) | (0 << OCIE0A) | (1 << TOIE0);

    UDIEN = 0; // USB interrupts are not handled - disable! //Glitch
    sei();     // allow interrupts globally //Set interrupts - Dont forget!!
}

/* Clear the 8 * 8 matrix */
void refresh_drawing()
{
    memset(drawing, 0, (sizeof(uint8_t) * 8 * 8));
}

/* Function fills framebuffer from the image (8 * 8 Matrix) - drawing */
void draw()
{
    uint8_t i, j = 0;

    for (i = 0; i < 8; i++)
    {
        framebuffer[i] = 0; //Clear the row for each drawing

        for (j = 0; j < 8; j++)
        {
            if (drawing[i][j] == 1) {
                framebuffer[i] |= (1 << (7 - j));
            }
        }
    }
}


int main(void)
{
    /* Initialize HW/interrupts  */
    init();
```

```
    while(1)
    {
        game_loop();
    }
}
```

6 References

- https://www.electronicwings.com/avr-atmega/atmega1632-external-hardware-interrupts#:~:text=Introduction,perform