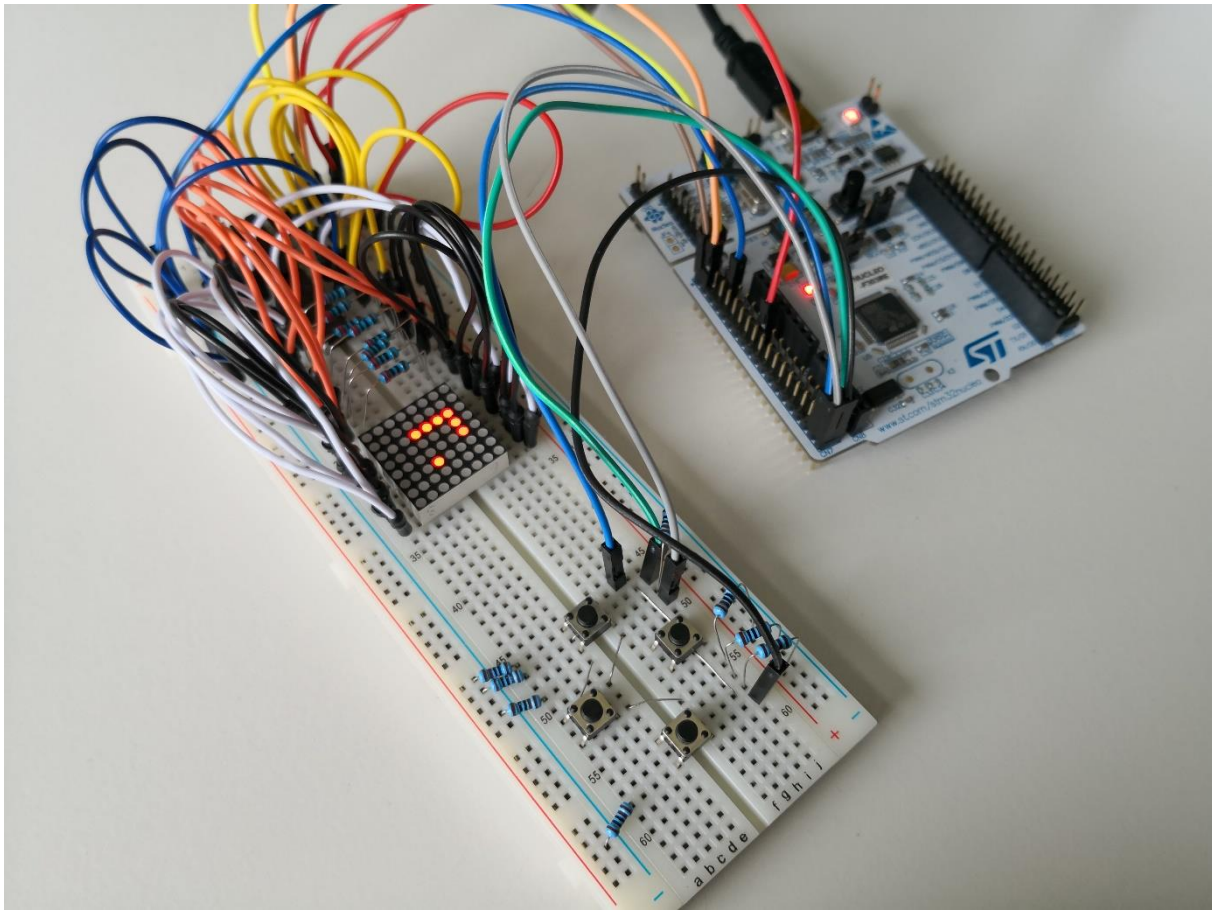
EXAMPLE PROJECT: EMBEDDED SNAKE GAME

Lucas Deutschmann



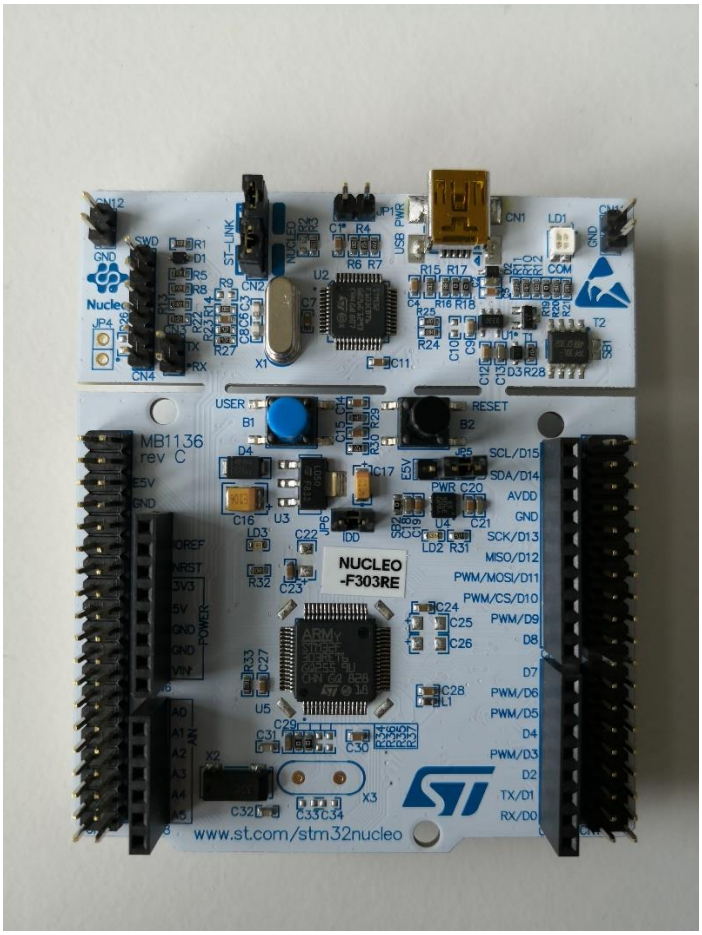
Inhaltsverzeichnis

1. Hardware

a) Microcontroller

The project is based on a STM32 Nucleo Board, which contains a STM32F303RET6 microcontroller. It provides the following features:

- ARM®32-bit Cortex®-M4 CPU
- 72 MHz max CPU frequency
- VDD from 2.0V to 3.6V
- 512 KB Flash
- 80 KB SRAM
- GPIO (51) with external interrupt capability
- 12-bit ADC (4)
- 12-bit DAC (2)
- Real-Time Clock
- Timers (10)

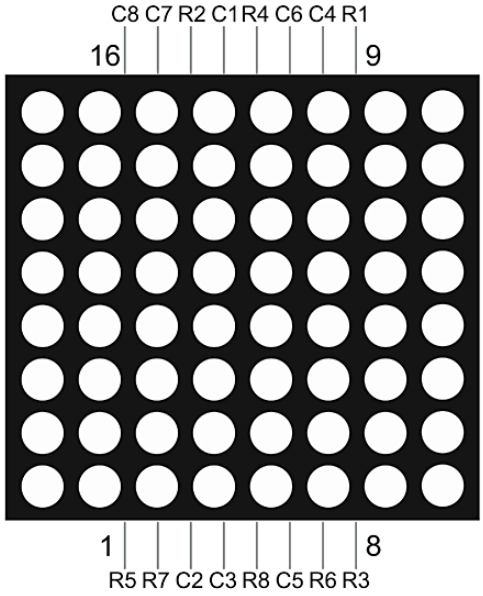


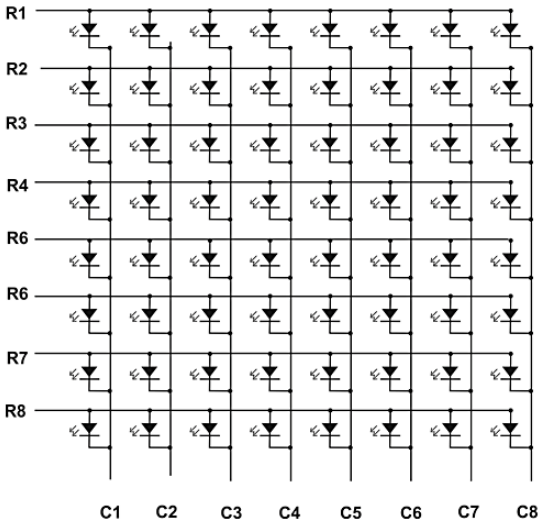
b) 8x8 LED Matrix

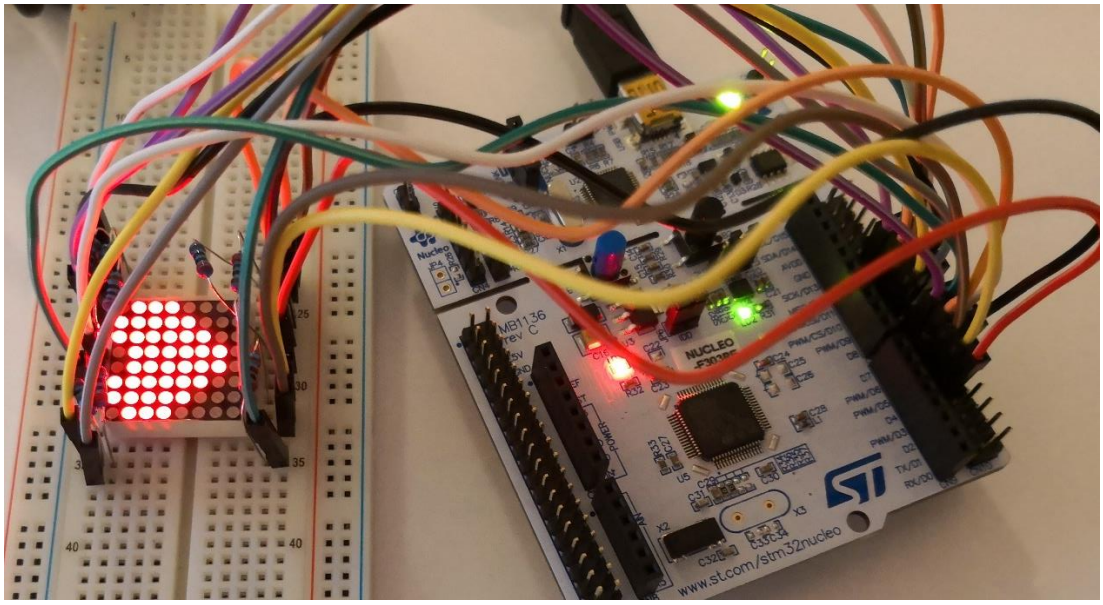
The game is displayed using an 8x8 LED Matrix. Each of the 16 pins in total corresponds to either a row or a column of the matrix. In order to light a specific LED, its row must be set to high and its column to low.

However, it is not possible to display arbitrary patterns by simply setting rows and columns. To be able to generate every possible pattern, a method called ***scanning*** is introduced. In scanning, we light every row sequentially. If the frequency of switching between different rows is high enough, it will appear as all LEDs are lit. In the project, a period of 100µs is used.

The LED matrix was tested in two applications. The first one simply used scanning to display a heart. The second application displayed the moving lettering “EMECSTHON 2019”. A video can also be found on Facebook. Both small projects required 16 GPIO pins of the Nucleo board.







c) 74HC595 Shift Register

In the previous section we introduced the LED matrix, which will be used as a display for the game. However, it uses 16 GPIO pins, which greatly reduces the possibility for extension of the project. To reduce the number of pins required, two 74HC595 shift registers are added.

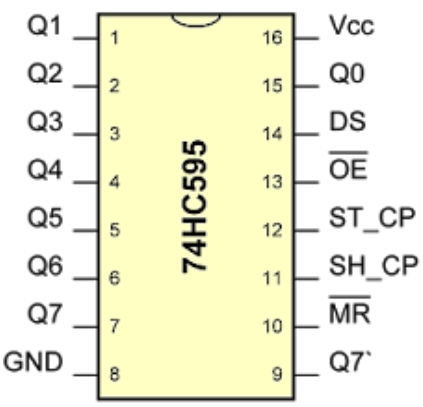


Figure 1: 74HC595 pinout diagram

Figure 1 shows the pinout diagram of the shift register. It can store 8 bits, which are provided sequentially on the DS port. At every positive clock edge of the SH_CP pin, data is shifted in and stored internally. With a positive edge at the ST_CP pin, the internal data is moved to the output registers Q0-Q7.

The shift registers can also be cascaded to increase their storage capacity. The Q7' pin provides the bit being shifted out. Hence, if we connect the Q7' of the first register with the DS pin of the second register, we create a 16-bit register. This is also done in the project, as we need 16 bits for controlling the LED matrix.

Using shift registers reduced the amount of GPIO pins we need from the board from 16 pins to 3 pins (not counting VDD and GND). Both registers share the SH_CP and ST_CP pin, as they function as a single 16-bit shift register. The third pin used is to provide the bits sequentially at the DS port.

2. Software

In this section some implementation details of the software are highlighted. The project has been created using the arm MBED online IDE.

a) Controlling the LED matrix

The current state of the LED Matrix is saved as an 8x8 bool array called *Field*. A true value corresponds to a lit LED with the same indices. This data is printed by the function *printField*, which is called inside an infinite loop.

```
55 void printField() {
56     for(unsigned int i = 0; i < 8; i++) {
57             for(unsigned int j = 0; j < 16; j++) {
58                 SH_CP = 0;
59                 if (i == j) {
60                     // Set active row
61                     DS = 1;
62                 } else if (j < 8) {
63                     // Deactivate other rows
64                     DS = 0;
65                 } else {
66                     // Set columns
67                     DS = !Field[i][j-8];
68                 }
69                 // Load new bit into shift register
70                 SH_CP = 1;
71             }
72             // Set output of shift register
73             ST_CP = 1;
74             ST_CP = 0;
75             wait(0.0001);
76     }
77 }
```

The function iterates over the rows i. For each row, 16 bits are provided to the shift registers via the DS pin. First, 8 bits corresponding to the one-hot encoded row are provided (j<8). Then, the actual field information for this specific row is shifted into the registers. At the end, a positive edge for the ST_CP port is given, outputting the new data to the matrix.

After each row, the function calls a wait of 100µs before addressing the next row. This time has been found empirically. At a switching period 1ms, the human eye was not able to see the scanning procedure any more. However, when filming the LEDs with a camera, the process was still recognizable.

b) The “Snake”-Class

Each part of the snake is represented by an object of a class named *snakeBody*. This class is structured like a linked list: It points to the previous as well as the next element of the snake. Furthermore, information about the position of the part is saved and can be modified by dedicated functions.

The head of the snake is of the class *snakeHead*, which is derived from snakeBody. It extends the base class by also saving the length of the snake and a pointer to the tail. Additionally, it provides a public function for appending new snakeBody elements and a function moving the entire snake into a given direction.

```
void move(Direction dir) {
    snakeBody * current = this->getTail();
    while(current != this) {
        // Move snake body parts forward
        current->setPos(current->getPrevPart()->getxPos(), current->getPrevPart()->getyPos());
        current = current->getPrevPart();
    }
    switch(dir) {
        // Move snake head to new position
        case UP: this->setyPos((this->getyPos()+1)%8); break;
        case LEFT: this->setxPos((this->getxPos()-1)%8); break;
        case RIGHT: this->setxPos((this->getxPos()+1)%8); break;
        case DOWN: this->setyPos((this->getyPos()-1)%8); break;
    }
}
```

When calling the move function, it iterates over all snakeBody elements, starting from the tail. Each element moves forward by updating its position to the position of the previous element. Lastly, the head updates its position based on the direction given as a parameter. Since we want to be able to move around the borders of our field, we perform this operation modulo 8.

c) External Interrupts

To be able to control our Snake, we will use four simple buttons, where each button corresponds to a direction. All GPIO pins are capable of enabling external interrupts. After initializing a certain pin as input, a function can be bound to it, which will be called every time the specified event occurs.

If a button gets pressed, the global variable for direction gets updated accordingly. However, only 90-degree turns are allowed.

d) Timed Interrupts

In addition to the external interrupt, a timed interrupt is used. The board provides 10 timers, which can be used to activate interrupts periodically. This functionality is used to call to move the snake and update the field.

```
 95 void move() {
 96     // Save tail position
 97     unsigned int xTail = snake->getTail()->getxPos();
 98     unsigned int yTail = snake->getTail()->getyPos();
 99     // Move snake
100     snake->move(dir);
101     if ((snake->getxPos() == xFood) && (snake->getyPos() == yFood)) {
102         // Snake has reached the food
103         snakeBody * newPart = new snakeBody(xTail, yTail);
104         snake->append(newPart);
105         Field[xTail][yTail] = 1;
106         spawnFood();
107     } else if (Field[snake->getxPos()][snake->getyPos()] == 1) {
108         // Snake has hit itself
109         NVIC_SystemReset();
110     } else {
111         // Snake moved to an empty field
112         Field[xTail][yTail] = 0;
113         Field[snake->getxPos()][snake->getyPos()] = 1;
114     }
115 }
116
```

The move function first saves the current position of the tail, as this LED will probably be turned off. Afterwards, we call the member function move of the snakeHead class and provide the direction the snake is supposed to move.

Depending on the position the head reaches, different actions are taken:

- If the snake reaches the position of the food, which is saved by *xFood* and *yFood*, a new snakeBody instance is created and appended to our snake. We also call the *spawnFood* function, which generates a new position for our food.
- In case the snake reaches a tile occupied by itself, we restart the game by simply resetting the device.
- Whenever the new position for the head is empty, we simply update the field accordingly.

By altering the timer interval, we can adjust the speed of the snake and hence modify the difficulty of the game.