

Variants 13-16 "Snake"

Write a control program for a game in which a snake (one LCD segment) moves across the entire LCD display. Control (up, down, right, left) – using the keyboard or switches.

	Controls	
Display Type	Keyboard	Switches
7-segment	Option 13	Option 15
2-line	Option 14	Option 16

EDSIM51 documentation

Please, refer to <https://edsim51.com/users-guide-2/> documentation to see details regarding **LCD Module, PIN Functions** and **LCD Module Instructions set**.

My implementation of SETUP_LCD uses the following instructions:

- Entry mode set
- Display off/on control
- Function set

SET_LCD_PTR_FROM_ACC uses the following instr:

- Set DDRAM address

Description of my Snake implementation on 8051

Reconnected LCD E, RS to P3.5 and P3.4

UART is needed to display the result of the game. Otherwise, the following interrupts will be used:

- EX1 (INT1) is an external interrupt 1 generated by the AND gate when one of the speakers on the keypad is 0-activated. To do this, you need to "AND gate enabled"
- Timer interrupt1 – read the keypad lines (one by one, hoping to get the interrupt above)
- **Interrupt by TIMER0:**
 - Updating the position of the snake (rewriting the coordinates)
 - Rearrange the snake segments to 50... 6Fh
 - Allow ADC conversion in order to find out how fast to update the game, i.e. what we will write to TH0 (then an interrupt is called for reading)
 - Updating the display
- A UART interrupt is called every time END_GAME when we transmit the next letter of the loss message
- An EX0 interrupt is required to read the ADC (the ADC must be enabled instead of the comparator!!)

At addresses 30h,...,4fh there are snake segments with *pointers to_the_position_on_the_the_ display* (the lower 5 bits of each cell 30..4f are responsible for the position on the display! from 00h to 1Fh), as well as artifacts of the food type (e.g., *display_position_pointer + 10000000b*):

- Segments should be placed sequentially!
- Food should follow immediately after the tail of the snake! (if, of course, there is food on the playing field)
- If several units of food are placed (only sequentially!), then their placement at addresses 30h,...,4fh should correspond to the sequence of meals, since each element of food becomes a segment of a snake **that must follow each other**. (Otherwise, a situation may arise when disparate elements of food become a snake after being consumed, but this violates the logic of the program, that is, the Player will see delirium until, presumably, he eats all the food)
Therefore, it is better not to place more than one element of food on the playing field.

At the addresses 50h,...,6Fh there are symbols in order for displaying on the display - using the FUNCTION_NAME function, we run through the addresses 30h,...,4Fh, take the *pointers_to_the_position_on_the_the_display*, and place the corresponding symbols (for example, H,s,t, M) in order. When updating the display, it is enough just to run through the addresses 50h,...,6Fh sequentially

Features of game mechanics

- Since the playing field is an ellipsoid with an area of 2 x 16, the movement of a two-segment snake vertically (the coordinates along this axis take two values - 0 and 1) is equivalent to the fact that the snake will begin to move "inside itself". Therefore, for reasons of isotropy of movement, we will not forbid the snake to move opposite to the initial impulse: for example, if it moves to the left, and then the player presses the "right" key, then if there are more than two segments, the snake will bump into itself, resulting in a GAMEOVER

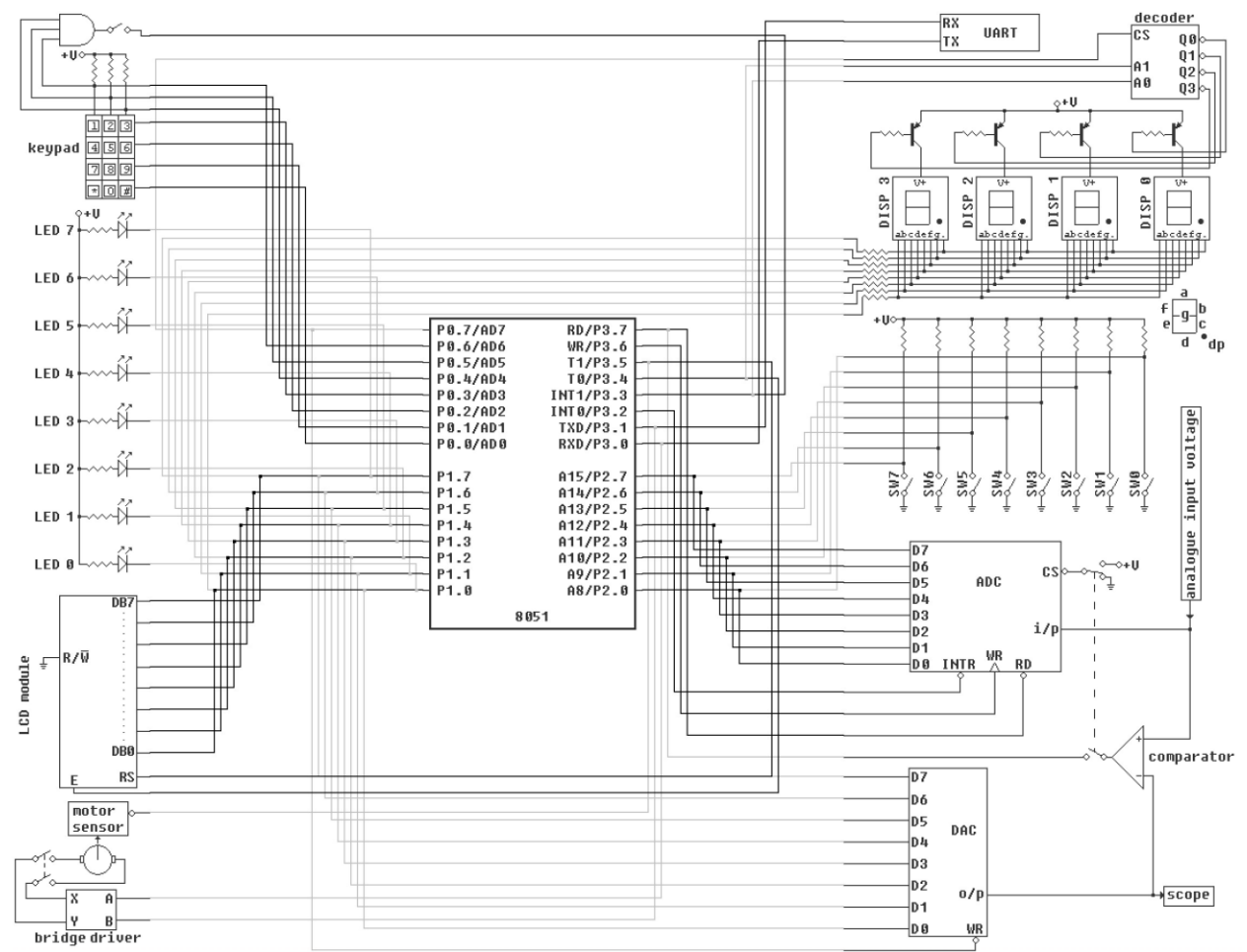
A piece of code that tested the admissibility of the opposite movement

```
CHECK_2ND_SEGMENT:
; if next_cell==2nd_segment, then we must use previous direction to move
    mov R1, #head_cell_addr
    mov A, @R1
    cjne A, tail_cell_addr, THERE_IS_2ND_SEGMENT
    jmp CHECK_SEGMENTS
THERE_IS_2ND_SEGMENT:
    inc R1
    mov A, @R1
    cjne A, 2h, CHECK_SEGMENTS
    mov A, key_reg
    cjne A, key_temp_addr, WRONG_KEY_1ST_TIME
        mov ENDGAME_TEXT_OFFSET_addr, #WRONGKEY_TEXT_OFFSET
        jmp CALL_END_GAME
WRONG_KEY_1ST_TIME:
    mov key_reg, key_temp_addr
    jmp FIND_NEXT_CELL

; if next cell != 2nd segment:
```

- When the Snake has filled the entire field, the game is considered won

Connections to the microcontroller are made as follows (wires are highlighted in black) (screenshot from EDSIM51):



CODE

```
; KEYPAD_INTS:
KEY_REG equ R7
KEY_TEMP_ADDR equ 7Fh
; UPDATE_CELLS:
RIGHT equ 6
LEFT equ 8
UP equ 10
DOWN equ 4
RESTART equ 9
BREAK equ 11

VERT_SIZE equ 2; should be power of 2
HOR_SIZE equ 16; should be power of 2

HEAD_CELL_ADDR equ 30h; only values between 00 and 1F are interesting
; if next cell is 0, than no snake's segment exists
TAIL_CELL_ADDR equ 5h; points to last non-zero byte in range (30h, 4fh)

TIMER0_SCALER_ADDR eq. 70h
TIMER0_SCALER equ 2

;-----
; UPDATE_CELLS_ADDRS and UPDATE_DISPLAY:
START_DISP_CELLS_ADDR eq. 30h
END_DISP_CELLS_ADDR equ 4fh

;-----
; UPDATE_DISPLAY:
FIRST_DISP_ROW_ADDR equ 0h
SECOND_DISP_ROW_ADDR equ 28h
;-----
; CLR_MEMORY:
clr_mem_start_addr eq. 71h
clr_mem_end_addr equ 72h
clr_mem_fill_with eq. 73h

;-----
KEYPAD_PORT equ P0

ADC_DATA_PORT equ P2
ADC_INTR_PIN equ P3.2
ADC_WR_PIN equ P3.6
ADC_RD_PIN equ P3.7

LCD_DB_PORT equ P1
LCD_RS_PIN equ P3.5
```

```

; P1.3
LCD_E_PIN equ P3.4
; P1.2

;-----
; END_GAME:
ENDGAME_TEXT_OFFSET_addr equ 7Eh
GAMEOVER_TEXT_OFFSET equ 0
CONGRATULATIONS_TEXT_OFFSET equ 11
WRONGKEY_TEXT_OFFSET equ 28
RESTART_TEXT_OFFSET equ 28
;-----
; ADD_MEAL:
MEAL_APPEARANCE_SPEED_SCALER_addr equ 7Dh
MEAL_APPEARANCE_SPEED_SCALER equ 10; <=> eqach 10th move
;-----
meal_ch equ 'X'
dietic_meal_ch equ 'x'
extra_cell_ch equ 'Y'
HEAD_ch equ 'H'
TAIL_ch equ 't'
SEGMENT_ch equ 's'

;-----

```

```

ORG 00h
    jmp START
ORG 3h; EXT0 trades => for ADC reading
    call EX0_INT_HANDLER_routine
    NETWORKS
ORG 0Bh; TIMER0 int. trades
    clr TR0
    ; setb PX1; <=> Ext.INT1 priority to handle interrupt inside Timer1 int.
    call T0_INT_routine
    ; clr PX1
    setb TR0
    GOALS; 6 bytes
org 1 p.m.; EXT1 Interruption
    clr tr1
    call EX1_INT_HANDLER_routine
    setb tr1
    NETWORKS

org 1Bh; TIMER1 int. trades
    ; clr TR1; INSIDE ROUTINE!
    ; setb PX1; INSIDE ROUTINE!
    ; <=> setb IP.2 <=> Ext.INT1 priority to handle interrupt inside Timer1 int.
    call T1_INT_routine
    ; clr PX1; at the end of routine

```

```

; setb TR1; at the end of routine
GOALS; 10 bytes
org 23h
clr TI
mov A, ENDGAME_TEXT_OFFSET_addr
movc A, @A+DPTR; move a byte from the code or program memory to A
mov SBUF, A
inc DPTR
NETWORKS

END_GAME_TEXT_ARRAY:
DB 'G','A','M','E',' ','O','V','E','R','!','#'
DB 'C','O','N','G','R','A','T','U','L','A','T','I','O','N','S','!','#'
DB 'W','R','O','N','G',' ','K','E','Y','#'
DB ' ','P','r','e','s','s',' ','R',' ','t','o',' ','r','e','s','t','a','r','t','#'
INIT_PROGRAM:
mov clr_mem_start_addr, #30h
mov clr_mem_end_addr, #6fh
mov clr_mem_fill_with, #0
call clr_memory

setb ea; enable interruptions

; timer0 used to update cells
call SETUP_TIMER0
call SETUP_LCD
;-----
call SETUP_KEYPAD
;-----
call SETUP_EX0; for ADC
RET
_RESTART:
mov TCON, #0
mov TMOD, #0
mov IE, #0
call INIT_PROGRAM
mov head_cell_addr, #0
mov tail_cell_addr, #0
mov key_reg, #right
mov key_temp_addr, #right
mov MEAL_APPEARANCE_SPEED_SCALER_addr, #MEAL_APPEARANCE_SPEED_SCALER
RET

START:
call INIT_PROGRAM
MAIN:

mov key_reg, #left
mov HEAD_CELL_ADDR, #01h
; mov 31h, #11h

```

```

; mov 32h, #12h
; mov 33h, #2h
; mov 34h, #3h
; mov 35h, #13h
; mov 36h, #14h
; mov 37h, #4h
mov tail_cell_addr, 30h
mov 31h, #10001111b; meal here at position 0Fh
mov MEAL_APPEARANCE_SPEED_SCALER_addr, #MEAL_APPEARANCE_SPEED_SCALER

RESTART_WAITING_LOOP:
    CJNE key_reg, #restart, $
    call _RESTART
    jmp RESTART_WAITING_LOOP

;-----
SETUP_TIMER0:
    mov timer0_scaler_addr, #timer0_scaler

    setb ET0; => enable Timer0 interrupt

    ; mov TMOD, #00000001b:
    ; TMOD.2=0 <=> T0/C0:=T0 mode
    ; TMOD.1=0 ;| <=> mode1 <=> 16-bit (256*256) mode
    ; TMOD.0=1 ;|
    ; TMOD.3=0; <=> GATE0=0 <=> INT0 cant turn on/off Timer0
    anl TMOD, #11110000b
    orl TMOD, #00000001b

    mov TH0, #00H;| Timer0 counts from 256*256-1
    mov TL0, #00H;|
    setb tr0; RUN TIMER0
    RET

TO_INT_routine:
    DJNZ timer0_scaler_addr, SKIP_UPDATING_CELLS
    call UPDATE_CELLS
    call UPDATE_CELLS_ADDRS
    call UPDATE_DISPLAY
    mov timer0_scaler_addr, #timer0_scaler

    djnz MEAL_APPEARANCE_SPEED_SCALER_addr, SKIP_UPDATING_CELLS
    call ADD_MEAL
    mov MEAL_APPEARANCE_SPEED_SCALER_addr, #MEAL_APPEARANCE_SPEED_SCALER

SKIP_UPDATING_CELLS:
    call CONTROL_GAME_SPEED

ret

```

UPDATE_CELLS:

CHECK_IF_NEXT_CELL_AVAILABLE:

FIND_NEXT_CELL:

mov A, head_cell_addr

;IF_KEY_REG_IS_RIGHT:

CJNE key_reg, #right, IF_KEY_REG_IS_LEFT

anl A, #00010000b;| Preserve green. coord

mov B, A ;|

mov A, head_cell_addr

INC A

IF_KEY_REG_IS_LEFT:

CJNE key_reg, #left, IF_KEY_REG_IS_UP

anl A, #00010000b;| preserve vert.coord

mov B, A ;|

mov A, head_cell_addr

DEC A

IF_KEY_REG_IS_UP:

CJNE key_reg, #up, IF_KEY_REG_IS_DOWN

SUBB A, #10h

anl A, #00010000b

mov B, A

mov A, head_cell_addr

IF_KEY_REG_IS_DOWN:

CJNE key_reg, #down, END_KEY_CASES

ADD A, #10h

anl A, #00010000b

mov B, A

mov A, head_cell_addr

END_KEY_CASES:

anl A, #00001111b; A:= A mod 16

add A, B

; NEXT CELL HERE:

mov 2h, A; 2h==R2 by default (if RS1==0 & RS0==0)

mov B, #0

CHECK_SEGMENTS:

mov r0, #head_cell_addr

; here we check if Snake bite itself (2h -> Head, A -> body segment:

CHECKING_SEGMENTS_LOOP:

mov A, @r0

CJNE A, tail_cell_addr, THERE_IS_NEXT_SEGMENT_TO_CHECK

; checking last segment:

; CJNE A, 2h, CHECK_EXTRA_CELLS; ; ; ; ; MOVING

; JMP CALL_END_GAME

; IF IT IS A TAIL, MOVING ALLOWED!!!

JMP CHECK_EXTRA_CELLS

THERE_IS_NEXT_SEGMENT_TO_CHECK:

inc r0; @R0 points to next Snake's segment if that exists

CJNE A, 2h, CHECKING_SEGMENTS_LOOP

```

; Snake will run onto itself!
mov ENDGAME_TEXT_OFFSET_addr,
#GAMEOVER_TEXT_OFFSET

JMP CALL_END_GAME

```

```

; 2h contains next cell Snake to move to (i.e. new head coord)
; here may be a meal for Snake:

```

```

CHECK_EXTRA_CELLS:

```

```

    inc r0
    ; let's check if it is truly an EXTRA cell:
    mov A, @r0
    anl A, #11100000b
    mov B, #0; reset B

```

```

    JZ CONTINUE_CHECKING_EXTRA_CELLS; MOVING; if not, we just ignore it

```

```

        mov B, r0; we will use B in case of eating
        mov A, @r0
        anl A, #00011111b; extract display position
        ; let's compare it with one stored in 2H
        ; clr CY
        subb A, 2h
        JZ EATING

```

```

CONTINUE_CHECKING_EXTRA_CELLS:

```

```

    CJNE r0, #4Fh, CHECK_EXTRA_CELLS
    JMP MOVING

```

```

EATING:

```

```

    mov r1, B
    mov A, @r1
    anl A, #11100000b
    add A, tail_cell_addr
    mov @r1, A
    jmp MOVING

```

```

; 2h contains next cell Snake to move to

```

```

MOVING:

```

```

; rewriting coord-s of Snake's segments

```

```

    mov r0, #head_cell_addr
    mov 3h, 2h; head_cell_addr; 2h contains new head pos

```

```

REWRITING_COORDs_LOOP:

```

```

    mov A, @r0
    mov @r0, 3h
    CJNE A, tail_cell_addr, THERE_IS_NEXT_SEGMENT_
        JMP END_REWRITING_COORDs_LOOP

```

```

THERE_IS_NEXT_SEGMENT_:

```

```

    mov 3h, A
    inc R0
    JMP REWRITING_COORDs_LOOP

```

```

END_REWRITING_COORDs_LOOP:
mov head_cell_addr, 2h
mov tail_cell_addr, 3h

; if there was a meal:
mov A, B; B contains meal position addr
JZ WAS_NO_MEAL; B should be in (31h, 4Fh)
mov r1, B
mov A, @r1
anl A, #11100000b
JZ WAS_NO_MEAL
    mov A, @r1
    anl A, #00011111b
    mov tail_cell_addr, A
    mov @r1, A
WAS_NO_MEAL:
RET

```

```

CALL_END_GAME:
call END_GAME

```

```
ret
```

```
;-----
```

```

UPDATE_CELLS_ADDRS:
; firstly let's clear all the bytes 50h,...,6fh
mov clr_mem_start_addr, #50h
mov clr_mem_end_addr, #70h
mov clr_mem_fill_with, #0
call clr_memory

```

```

mov A, head_cell_addr
anl A, #00011111b; A := A mod 32 - obtaining cell's display addr
add A, #50h; ; indent
mov R0, A
mov @R0, #head_ch; H for head

```

```

mov R1, #head_cell_addr
inc R1
; check if Head==Tail
mov A, tail_cell_addr
subb A, head_cell_addr
JZ UPDATING_EXTRA_CELLS_ADDRS_LOOP

```

```

UPDATING_CELLS_ADDRS_LOOP:
mov A, @R1
anl A, #00011111b

```

```

add A, #50h
mov R0, A

mov A, @R1
CJNE A, tail_cell_addr, CONTINUE_UPDATING_CELLS_ADDRS_LOOP
    mov @R0, #tail_ch; t for tail
    JMP UPDATING_EXTRA_CELLS_ADDRS_LOOP

```

```

CONTINUE_UPDATING_CELLS_ADDRS_LOOP:
mov @R0, #segment_ch; s for segment
inc R1
jmp UPDATING_CELLS_ADDRS_LOOP

```

; let's update such cells as MEAL and so on:
; EACH ZERO-CELL here supposed to mean nothing!

```

UPDATING_EXTRA_CELLS_ADDRS_LOOP:
    mov A, @R1
    ; JZ UPDATING_EXTRA_CELLS_ADDRS_LOOP
    anl A, #00011111b
    add A, #50h
    mov R0, A; now R0 contains addr under consideration

```

```

    mov A, @R1
    anl A, #11100000b
    ;IF_CELL_IS_EXTRA1:
    cjne A,#10000000b, IF_CELL_IS_EXTRA2
        mov @R0, #meal_ch
        jmp IF_CELL_EMPTY
    IF_CELL_IS_EXTRA2:
    cjne A,#01000000b, IF_CELL_IS_EXTRA3
        mov @R0, #'x'; dietic_meal_ch
        jmp IF_CELL_EMPTY
    IF_CELL_IS_EXTRA3:
    cjne A,#00100000b, IF_CELL_EMPTY
        mov @R0, #'y'; extra_cell_ch
        jmp IF_CELL_EMPTY
    IF_CELL_EMPTY:
    inc R1
    CJNE R1, #4Fh, UPDATING_EXTRA_CELLS_ADDRS_LOOP
; EXIT_UPDATING_CELLS_ADDRS:
RET

```

;-----

```

SETUP_LCD:
    clr LCD_RS_pin; P1.3          ; to send data to LCDs [IR]
    ; function set:
    mov LCD_DB_port, #00111000b; FUNCTION SET:
        ; LCD_DB.5=1 necessarily, LCD_DB.4=1 for 8bit mode, LCD_DB.3=1 for 2 lines,
        LCD_DB.2=0 for 5x8 dots font

```

```
setb LCD_E_pin; P1.2;| negative edge on E
clr LCD_E_pin; P1.2 ;|
```

```
call DELAY; wait for BF to clear
```

```
; entry mode set:
mov LCD_DB_port, #00000110b; inc move, no shift
```

```
setb LCD_E_pin;| negative edge on E
clr LCD_E_pin ;|
```

```
call DELAY; wait for BF to clear
```

```
; display on/off control
mov LCD_DB_port, #00001111b; enable display, cursor and blinking
```

```
setb LCD_E_pin;| negative edge on E
clr LCD_E_pin ;|
```

```
call DELAY; wait for BF to clear
```

```
setb LCD_RS_pin; to send data to LCDs [DR]
RET
```

SET_LCD_PTR_FROM_ACC:

```
add A, #80H          ; this command starts with 1: 1XXX_XXXX
clr LCD_RS_pin       ; to send data to LCDs [IR]
mov LCD_DB_port, A    ; send _CMD to LCDs [IR]
setb LCD_E_pin        ;| negative edge on E
clr LCD_E_pin         ;|
call DELAY
setb LCD_RS_pin       ; to send data to LCDs [DR]
RET
```

UPDATE_DISPLAY:

```
mov R0, #4fh
mov A, #first_disp_row_addr
call SET_LCD_PTR_FROM_ACC
UPDATING_1_DISP_ROW_LOOP:
    inc R0
    mov A, @R0
    call SEND_CHARACTER
    cjne R0, #60h, UPDATING_1_DISP_ROW_LOOP
dec R0
mov A, #second_disp_row_addr
call SET_LCD_PTR_FROM_ACC
UPDATING_2_DISP_ROW_LOOP:
    inc R0
```

```

    mov A, @R0
    call SEND_CHARACTER
    cjne R0, #70h, UPDATING_2_DISP_ROW_LOOP

```

```

RET

```

```

SEND_CHARACTER:

```

```

    mov LCD_DB_port, A          ; data from A -> LCD
    setb LCD_E_pin              ;| negative edge on E
    clr LCD_E_pin               ;|
    call DELAY                  ; wait for BF to clear
    RET

```

```

;-----

```

```

; KEYPAD_INTS:

```

```

SETUP_KEYPAD:

```

```

    ; SETTING_UP_EXT1:
    setb EX1; set EX1 (==IE.2) to enable ext. interruption INT1
    ; setb IT1; INTs by front;
    ; setb IT1 if we want to process pressing key for only 1 time
    ; doesnt work

```

```

    call SETUP_TIMER1
    RET

```

```

SETUP_TIMER1:

```

```

    setb ET1; => enable Timer1 interrupt
    ; mov TMOD, #00100000b:
    ; TMOD.6=0; <=> T1/C1:=T1 mode
    ; TMOD.5=1;| mode2 <=> 8-bit auto-reload mode
    ; TMOD.4=0 ;|
    ; TMOD.7=0; <=> GATE1=0 <=> INT1 cant turn on/off Timer1
    anl TMOD, #00001111b
    orl TMOD, #00100000b

```

```

    mov TH1, #80H;
    mov TL1, #80H;

```

```

    setb tr1; RUN TIMER1
    RET

```

```

T1_INT_routine:

```

```

    clr TR1
    setb PX1; set EXT1 int priority

```

```

    mov key_temp_addr, #0

```

```

    setb KEYPAD_port.3

```

```
clr KEYPAD_port.0
nop
mov key_temp_addr, #3
```

```
setb KEYPAD_port.0
clr KEYPAD_port.1
nop
mov key_temp_addr, #6
```

```
setb KEYPAD_port.1
clr KEYPAD_port.2
nop
mov key_temp_addr, #9
```

```
setb KEYPAD_port.2
clr KEYPAD_port.3
nop
```

```
mov R0, #0
jnb F0, RET_T1_INT_HANDLER_routine
    clr F0
```

```
RET_T1_INT_HANDLER_routine:
    mov key_temp_addr, key_reg; here we will contain previous key state
    setb tr1
    clr px1
```

```
RET
```

```
EX1_INT_HANDLER_routine:
    MOV A, key_temp_addr
    JB KEYPAD_port.4, if_nCol0
        mov key_reg, A
        setb F0
    if_nCol0:
        inc A
    JB KEYPAD_port.5, if_nCol1
        mov key_reg, A
        setb F0
    if_nCol1:
        inc A
    JB KEYPAD_port.6, int_end
        mov key_reg, A
        setb F0
    int_end:
        mov KEYPAD_port, #0xFF
        clr ie1
    RET
```

```
;-----
```

; END_GAME:

SETUP_UART:

```
    mov IE, #10010000b
        ;| |
        ;| enable UART int-s
        ; reset all int-s blocking
```

```
    clr SM0 ;| UART to mode 1
    setb SM1;|
```

```
    mov A, PCON;| SMOD -> double baud rate
    setb ACC.7 ;|
    mov PCON, A;|
```

```
    mov TMOD, #20H      ; T1 to mode 2 (auto-reload)
    mov TH1, #-3        ;| Setup T1 to work with 19200 baud UART
    mov TL1, #-3        ;|
    RET
```

END_GAME:

```
    setb PS; UART int has high priority
```

```
    call SETUP_UART
    setb tr1
```

```
    mov DPH, #high(END_GAME_TEXT_ARRAY)
    mov DPL, #low(END_GAME_TEXT_ARRAY)
```

```
    setb TI; once
    nop
```

```
UART_RESULT_TEXT_LOOP:
    cjne A, #'#', UART_RESULT_TEXT_LOOP
```

```
    mov A, #0
    mov ENDGAME_TEXT_OFFSET_addr, #RESTART_TEXT_OFFSET
UART_RESTART_TEXT_LOOP:
    cjne A, #'#', UART_RESTART_TEXT_LOOP
```

```
; END_UART_TRANSMISSION:
    clr PS; reset UART int priority
    call SETUP_KEYPAD
    RET
```

;-----

; the code is almost exact as the code from LW6.task1 example
; HERE WE CONTROL GAME SPEED WITH ADC:

SETUP_EX0:

```
    setb IT0; EX0 is edge-activated
    setb EX0; enable
    RET
```

```

EX0_INT_HANDLER_routine;; responds to ADC conversion interrupt
    clr ADC_RD_pin; clr ADC data lines => enable data lines
    mov TH0, ADC_DATA_port; take data from ADC
    setb ADC_RD_pin; disable data lines
    RET

```

```

CONTROL_GAME_SPEED:
    clr ADC_WR_pin;|
    setb ADC_WR_pin;| positive edge to start ADC conversion
    RET

```

```

;-----

```

```

ADD_MEAL:
    mov R0, #head_cell_addr
CHECKING_CELLS_LOOP:
    mov A, @R0
    inc R0
    cjne A, tail_cell_addr, CHECKING_CELLS_LOOP
        ; R0 points to first not-segment cell
    cjne R0, #50h, CHECK_CELL_FOR_MEAL
        ; here we found out, that no free cells available => GAME WON!
        mov ENDGAME_TEXT_OFFSET_addr,
#CONGRATULATIONS_TEXT_OFFSET
        jmp CALL_GAME_END_
CHECK_CELL_FOR_MEAL:
    mov A, @R0
    anl A, #10000000b
    JZ NEW_CELL_AVAILABLE
        ; meal already exists:
        RIGHT; do nothing
NEW_CELL_AVAILABLE:
    mov R1, #50h
    SEARCHING_1ST_FREE_ADDR_LOOP:
        mov A, @R1
        ; we may also check A for being some other cell type:
        ; 010xxxxxb or 001xxxxxb, but will do not, supposing, we have
not such type cells)
        inc R1
        jnz SEARCHING_1ST_FREE_ADDR_LOOP
            ; We have found an empty addr (contained in R1--):
            dec R1
            mov @R1, #meal_ch
            mov A, R1
            clr CY
            subb A, #50h
            orl A, #10000000b
            mov @R0, A

    RET

```

```
CALL_GAME_END_:  
    call END_GAME  
; some service functions:
```

```
delay:  
    MOV R6, #50          ; just makes program to wait 50 cycles  
    DJNZ R6, $  
    RET
```

```
CLR_MEMORY:  
    mov A, clr_mem_start_addr  
clr_mem_loop:  
    mov r0, A  
    mov @r0, clr_mem_fill_with  
    inc A  
    cjne A, clr_mem_end_addr, clr_mem_loop  
    RET  
ret
```