Microprocessor Course Product Report – Snake Game

David Ragusa 9/3/15

Abstract

The famous game Snake was programmed on the Atmel 128 microcontroller. The XY mode on an oscilloscope was used as the screen, and for this two digital digital-to-analogue converters were constructed on the breadboard. Input was taken from a 4x4 keyboard, and menus and text output were displayed on the LCD display. In addition, persistent high scores were implemented, using the microcontroller's EEPROM, and sound effects were added to the game by using the onboard timer and interrupts.

Introduction

Snake as a game concept has been around at least for 40 years.[4] In this project, the classic Nokia version was chosen, where the snake moves in the cardinal directions on a 2D grid. Snake was chosen as the project because it was thought that it had the potential to utilise a wide array of tools and programming, as well as some electronics. Among the features to be used were random number generation to place the snake food, an extensive system of menus to set game options and view highscores, hexadecimal to decimal number conversion for score display on the LCD, EEPROM I.O. for persistent high scores, using the timer and interrupts to generate sound effects, and some electronics for the oscilloscope display.

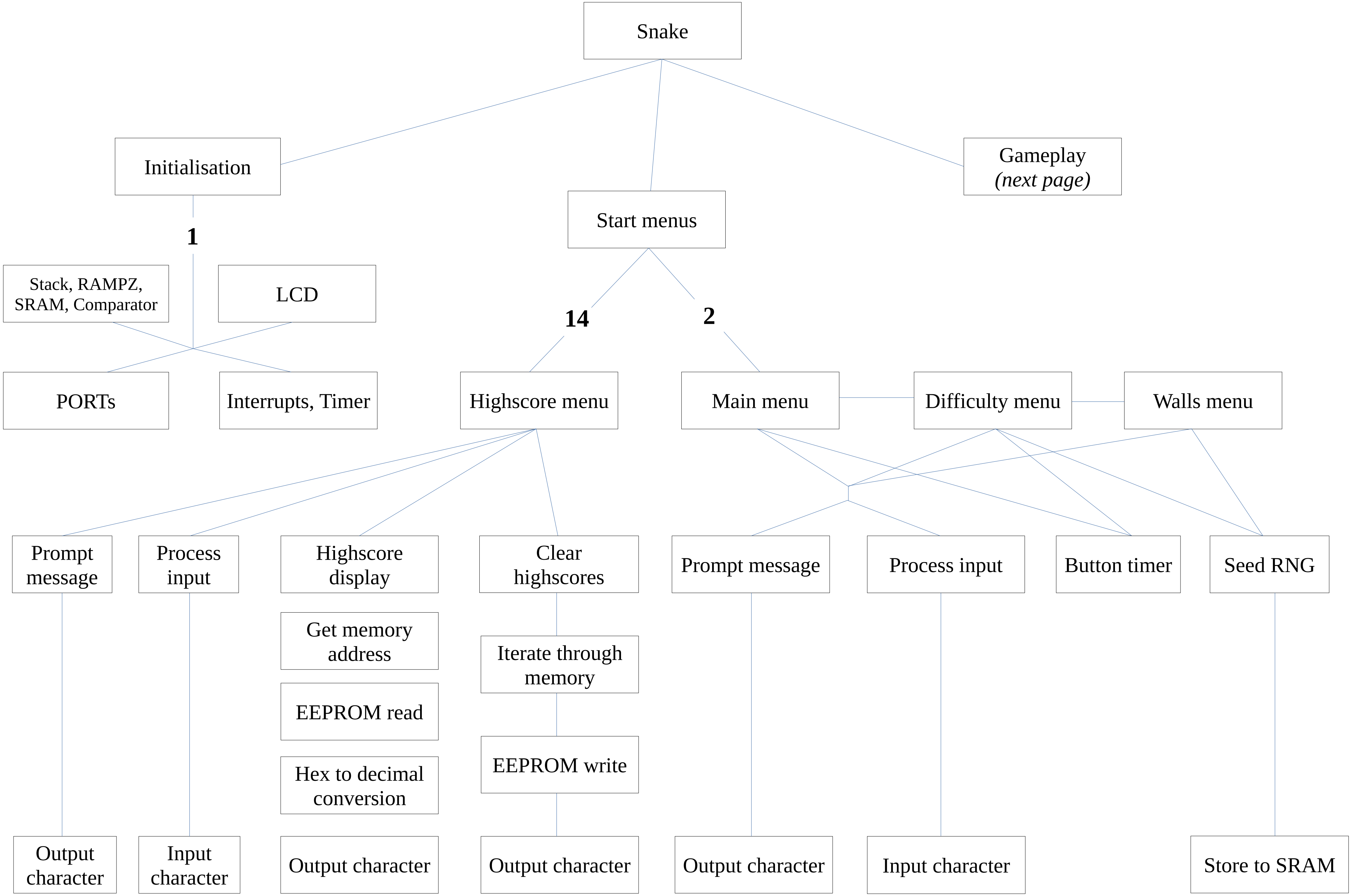
High Level Design

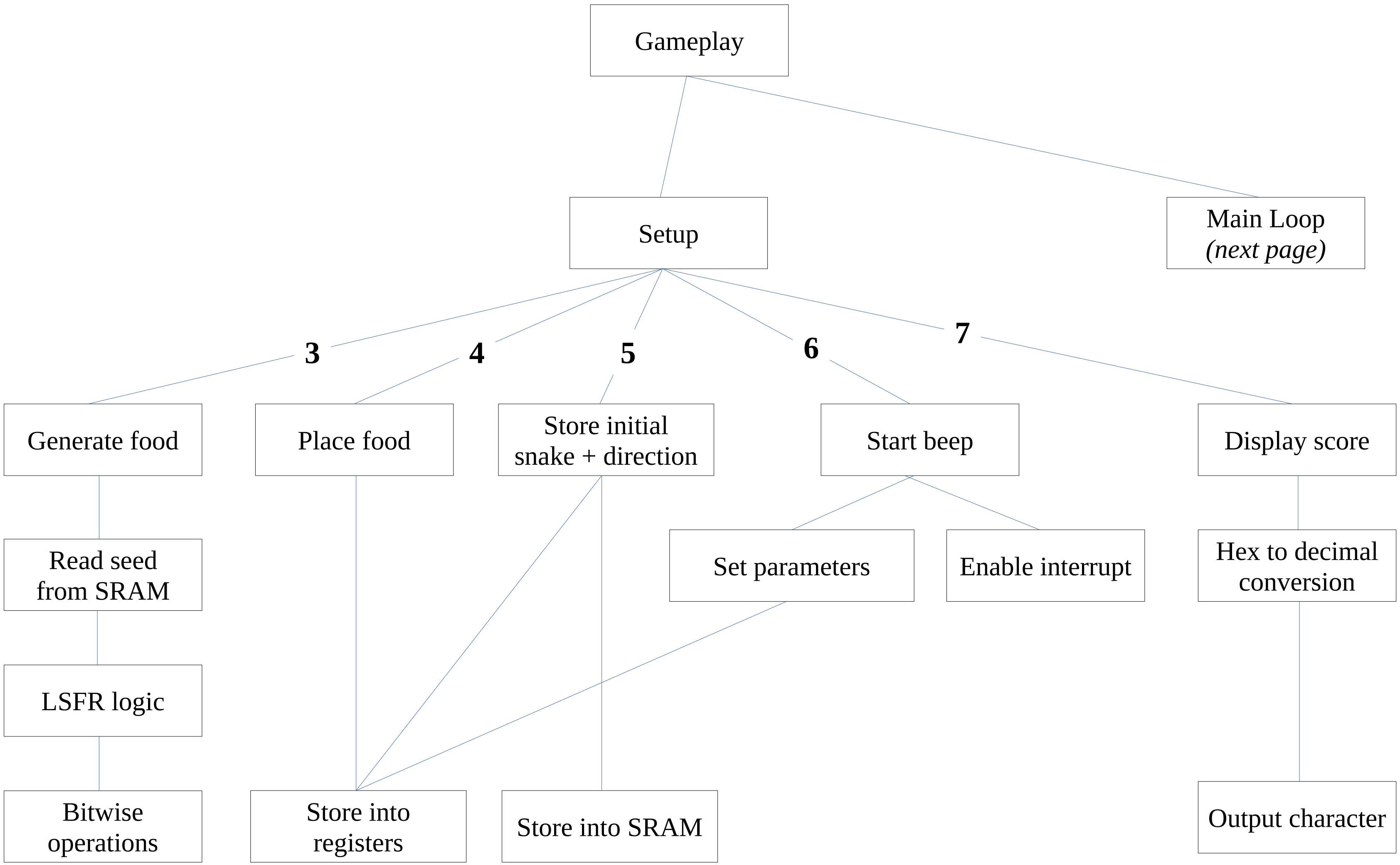
The program is divided into several modular sections, represented on the next four pages. Individual sections are referred to by bold numbers on the diagrams and are described in much greater detail further on in the report. The diagrams are broadly organised with time increasing from left to right and complexity increasing from bottom to top.

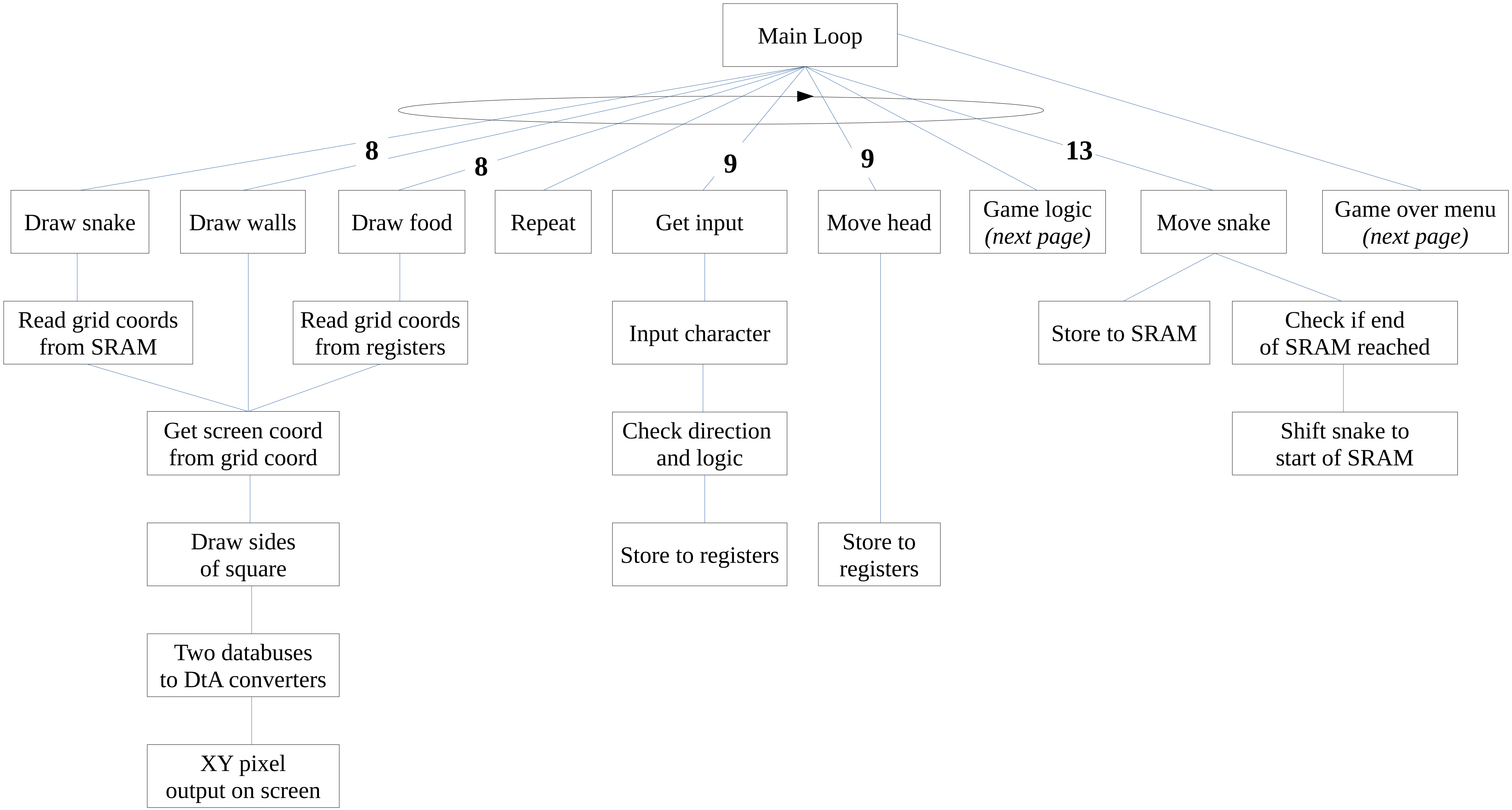
At the highest level, there is general initialisation, which sets up everything needed for the program to function. There is also the menu system, which uses the LCD as output and the 4x4 keyboard as input, and allows the user to view and manage highscores, as well as set options for the next game. Finally, there is the gameplay section.

Within gameplay, there is the individual game setup and the main gameplay loop. This draws the snake, food and walls to the oscilloscope display via two digital-to-analogue converters on the breadboard. Drawing is repeated a number of times, depending on the difficulty chosen by the player – the more drawing is repeated, the easier the game. A game logic step is triggered when the drawing loop has finished.

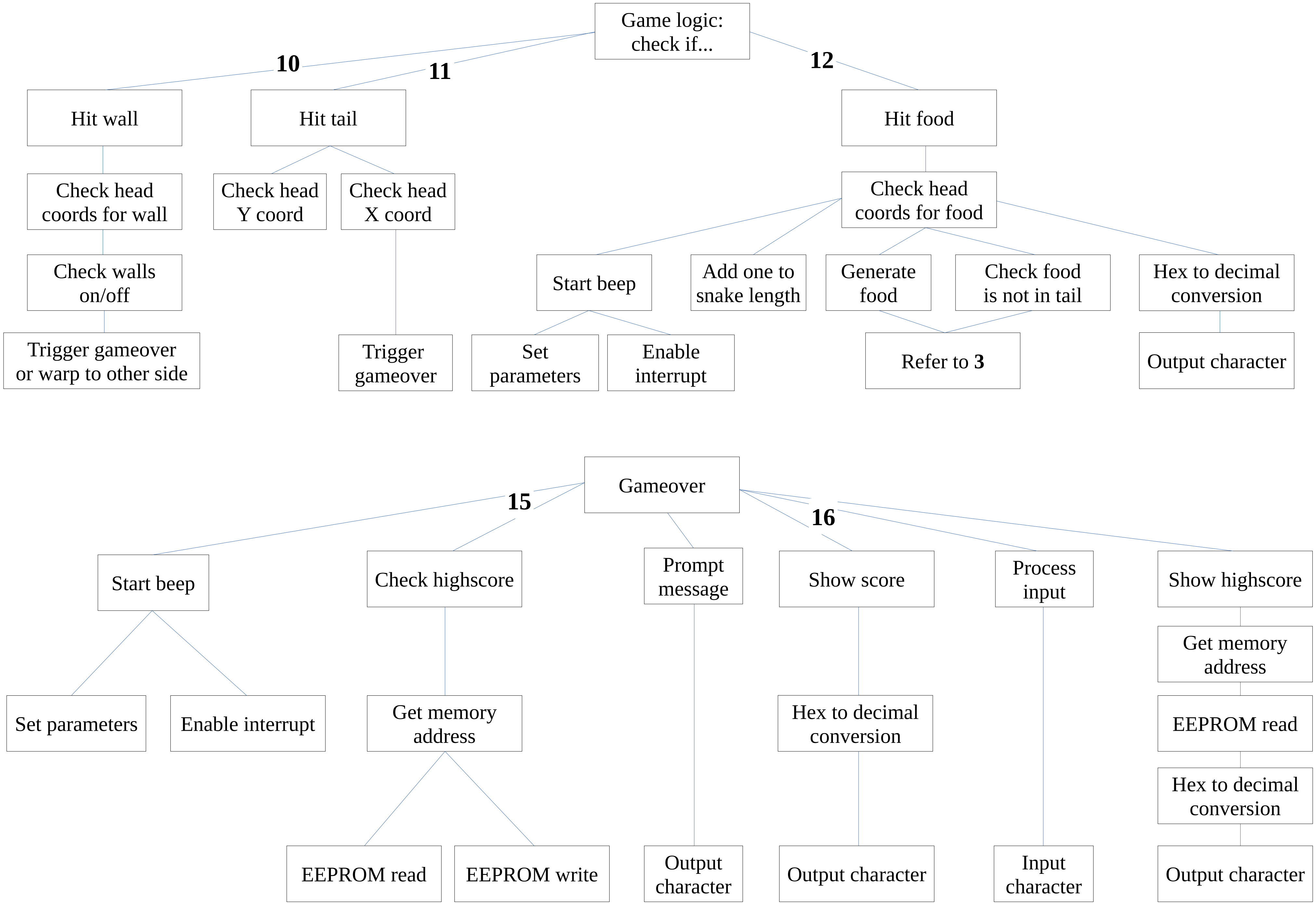
(cont. after the modular design pages)


Snake
Initialisation
Start menus
Gameplay
(next page)
1
Stack, RAMPZ, SRAM, Comparator
LCD
PORTs
Interrupts, Timer
14
2
Highscore menu
Main menu
Difficulty menu
Walls menu
Prompt message
Process input
Highscore display
Clear highscores
Prompt message
Process input
Button timer
Seed RNG
Get memory address
EEPROM read
Hex to decimal conversion
Iterate through memory
EEPROM write
Output character
Input character
Output character
Output character
Output character
Input character
Store to SRAM


Gameplay
Setup
Main Loop
(next page)
3
4
5
6
7
Generate food
Place food
Store initial snake + direction
Start beep
Display score
Read seed from SRAM
Set parameters
Enable interrupt
Hex to decimal conversion
LSFR logic
Bitwise operations
Store into registers
Store into SRAM
Output character



Main Loop
8
8
9
9
13
Draw snake
Draw walls
Draw food
Repeat
Get input
Move head
Game logic
(next page)
Move snake
Game over menu
(next page)
Read grid coords
from SRAM
Read grid coords
from registers
Input character
Store to SRAM
Check if end
of SRAM reached
Get screen coord
from grid coord
Check direction
and logic
Shift snake to
start of SRAM
Draw sides
of square
Store to registers
Store to
registers
Two databuses
to DtA converters
XY pixel
output on screen


Game logic: check if...
10
11
12
Hit wall
Hit tail
Hit food
Check head coords for wall
Check head Y coord
Check head X coord
Check head coords for food
Check walls on/off
Trigger gameover or warp to other side
Trigger gameover
Start beep
Add one to snake length
Generate food
Check food is not in tail
Hex to decimal conversion
Set parameters
Enable interrupt
Refer to 3
Output character
Gameover
15
16
Start beep
Check highscore
Prompt message
Show score
Process input
Show highscore
Get memory address
Set parameters
Enable interrupt
Get memory address
Hex to decimal conversion
EEPROM read
Hex to decimal conversion
EEPROM read
EEPROM write
Output character
Output character
Input character
Output character

Within game logic, input is taken from the 4x4 keyboard. This is checked for validity, and then the head of the snake is moved. Based on the new position of the head, the game checks if the snake has hit a wall, its own tail or a piece of food. If the snake has hit a wall, behaviour depends on the wall setting chosen by the player. If the snake has hit its own tail, game over is triggered. If the snake has hit a piece of food, the food beep is triggered, the length of the snake is increased by one, and a new food is generated, making sure that it does not lie within the tail of the snake.

If game over has not been triggered, the rest of the snake is moved. This consists of storing the new head position in SRAM. If, however, this approaches an area of SRAM that cannot be used (`$0E00`, possibly encroaching on the stack), then a routine is triggered which moves the entirety of the snake to the start of available SRAM.

If there is a game over instead, several things happen: the game over beep is triggered, the player's score is checked against the high score and overwrites it as necessary, and a menu is then shown displaying the player's score and the high score. The player then has the option to either replay a game with the same settings as the current one, or go back to the initial menu.

Software and Hardware Design

1) Initialisation and setup

Aside from the boilerplate setup code for the stack, RAMPZ, SRAM, LCD, and comparator, of note is that all I/O ports are used. Ports A and C are used for the LCD, ports B and D are used for the y and x databuses respectively, port E is used for the 4x4 keyboard input, and port F is used for sound output. Annoyingly, port F was out of range for the `out` command, so the `sts` command had to be used to set the direction register and output data.

Interrupts were enabled for the sound effects. A very simple sound system was implemented using the microcontroller's onboard timer 0, set to 'clear on compare match'. This means that the timer counter (`TCNT0`) constantly increases, and when it hits a specified value (`OCR0`), the counter is reset and the timer 0 compare interrupt is triggered (`Timer0comp`). In the interrupt function, an on/off value was inverted at every function call, producing a square wave on the output at port F. A word (`freqdur`) was used as the counter for how many times the interrupt would run before disabling itself. Hence `OCR0` and `freqdur` control the frequency and duration of a square wave output, respectively.

Another two values provided the ability to play additional notes. If `notecount` was more than zero, the interrupt function did not disable itself, but loaded new parameters into `OCR0` and `freqdur`, changing the note. The note changed to was controlled by `soundselect`.

It was decided that the actual sounds produced were of minor importance, and so trial and error was employed to get the beep to a satisfactory pitch and duration. Headphones were plugged into the sound output on the breadboard, but the volume was far too loud. To solve this, a simple potential divider was used to bring the voltage, and therefore volume, down to an acceptable level (fig. 1).

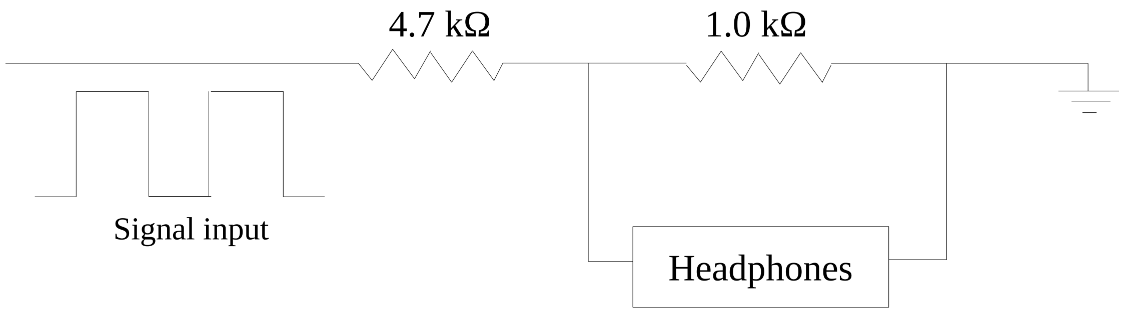


Fig. 1: Potential divider to reduce sound volume

2) Menu system

In all the menus, a splash screen is constantly displayed (`Splashscreen`, fig. 2). This has aesthetic value as well as allowing the user to centre the display. For the method of output of this screen, refer to section **8**. This also illustrates a key design point – the game grid. The screen is subdivided into squares, each 8 pixels wide. All game logic and display takes place at the game grid level, effectively giving us a 31x25 display.

Fig. 2: Splash screen

Menus all output text to the LCD and accept input from the 4x4 keyboard, so those processes will be explained here. Messages are stored in byte tables in program memory, and concatenated with the non-printable character 0. When text is displayed on the LCD, the byte table is loaded to the `Z` address, and the `MessMore` routine puts characters out to the LCD one at a time until 0 is reached. In this way, hardcoding of each message length can be avoided.

To accept input from the 4x4 keyboard (`Buttonpress`, the column and row of the button have to be read separately. They are combined into a single byte, which is stored in `TempReg`. Values for the numbers and characters were calculated separately, which allowed `cpi` comparisons for conditionals in the program.

The main menu leads either to the highscore menu (section **14**), or to the game options – difficulty and walls menus. Difficulty is saved in `TempReg`, ranging from `$12` for easy to

`$04` for hard, and the wall preference is saved in `wallsreg`, where `$FF` is walls on and `$00` is walls off.

Of note here is the method of obtaining random seeds. It is important that the seeds be different at every initialisation, otherwise the food positions would be the same each time the microcontroller was reset. Usually an open analog pin would be used as a source of randomness, but the Atmel pins are not sensitive enough for such a task.[1] To solve the problem, a counter was added to the main and difficulty menus. This would constantly increment (rolling over to `$00` once it reached `$FF` of course), and would be used as the seed once the next menu was reached. Hence the time taken for the player to select an option is the source of randomness – seeing as the microcontroller speed is far superior to the speed of button presses.

3) Food generation

An 8 bit linear feedback shift generator (LFSR) was implemented, in a Galois configuration, to provide a source of random numbers. The Galois configuration was chosen over Fibonacci because it works on the whole byte instead of having to execute bit by bit operations, and is therefore faster.

A diagram of the working of the LFSR is shown in fig. 3. The choice of 'tap' bits is important in order to provide coverage over all byte values. To this end, the 1st, 5th, 6th, and 8th bits were tapped.[2] If the 1st bit (the output bit) is a 0, every bit is shifted to the right, inserting a 0 at the left. If the output bit is a 1, the tapped bits are inverted, and every bit is shifted to the right, inserting a 1 at the left.

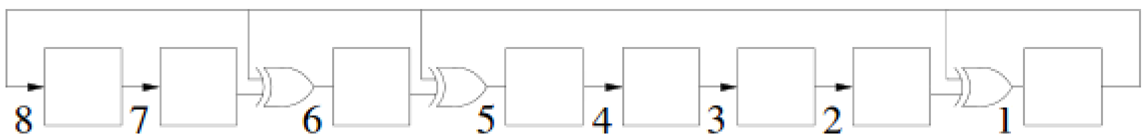


Fig. 3: 8-bit Galois LFSR with taps at 1,5,6,8 [5]

The `Randomfood` routine retrieves the x and y seeds from SRAM, runs the LFSR on each, and stores the results back to SRAM as the new seeds. It returns the results, modulo the size of the game area (29 for x, 23 for y). The first seeds are obtained from the menus (section 2).

4) Food placement

`Randomfood` also increments the x and y coords by one to avoid the food spawning in the wall – seeing as modulo arithmetic can subtract to 0, which is a wall coord. They are then placed in separate registers to avoid constantly pushing/popping them, and for easier comparisons.

5) Initial snake and direction

The snake is represented by four values in the program: `X`, the snake length, `Y`, the location in SRAM of the first snake segment, and `headX` and `headY`, the x and y coords respectively of the head of the snake on the game grid. When the snake grows in length, `headX` and `headY` are appended to the `Y` location. Hence the entire snake can be displayed simply by counting `X` squares down from `Y` in SRAM. `buttonstate` represents the current direction of the snake.

In `Gamesetup`, the snake is placed at the top left of the screen, with an initial direction going right, and a length of 10.

6) Start beep

This produces a two-note rising beep – explained in section **1**.

7) Score display

In order to display the score on the LCD, 10 is first subtracted from `X` (the initial snake length doesn't count!) and then `X` is converted to a three digit decimal number. `Decimaloutput` achieves this by counting and subtracting the number of 100s present, then the 10s, and finally the 1s. A byte table of decimal numbers is supplied, so that the correct text character can be output by simply counting along this table however many times `Decimaloutput` found for each digit (`MessDecChar`).

8) Draw cycle

The snake, walls and food are all output to screen in a similar manner: they are all a collection of x and y coords on the 31x25 game grid. To display the snake, `X` number of x and y coords are obtained by counting down from `Y`. To display the walls, the coords are obtained by starting at 0, 0 (top right), and going round the edges of the grid anticlockwise. The food is displayed by outputting `foodX` and `foodY` directly. The splash and game over screens are lists of x and y coords that are directly output (e.g. `splashdata`).

Once a game grid coord has been obtained, it is multiplied by 8 to get to the correct pixel location – seeing as the grid squares are 8 pixels wide. The grid square is then drawn by going round the square anticlockwise, 4 pixels output per side. This all takes place in `Squaredisplay`. Seeing as the oscilloscope naturally draws by vector graphics (the electron beam has to physically sweep across to a new location), each square is drawn twice to maximise the intensity of the desired output. In this way the brightness of the screen can be lowered so the stray artifacts of the display are much less visible.

The actual oscilloscope display was obtained by setting the scope to XY mode. In this mode, two analog inputs correspond to a pixel position on the screen.

The analog inputs to the oscilloscope was obtained by running the databuses, ports D and B, to two digital-to-analogue converters as shown in fig. 4 (`Outapixel`). As data is sent in parallel, the write enable pin can be kept grounded – write is always on.

After some preliminary experimentation, it was found that the maximum number of pixels obtainable while keeping the coordinates small enough to fit in a byte, was ≈ 250 by 200. Hence the choice of a 31x25 game grid (248x200 pixels).

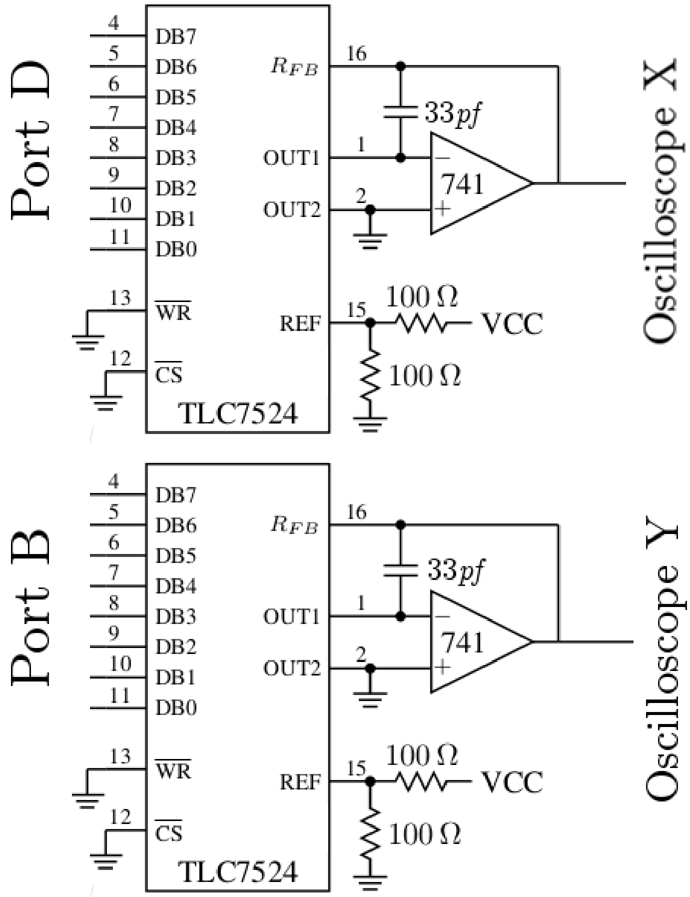


Fig. 4: Circuit diagram for the DtA converters [3]

9) Input and moving the head

Input is taken from the 4x4 keyboard (section 2). First, the program checks that the byte received is a direction (2, 4, 6, or 8). If it is, it then checks if the direction is opposite to the current direction (`Directionpress`) – the snake cannot go back on itself! Only if the button passes these two conditions is `buttonstate` updated.

`Movehead` very simply changes the `headX` or `headY` coords based upon the value of `buttonstate`. This is necessary because all the following game logic relies on `headX` and `headY`.

10) Game logic – hitting a wall

Detection of hitting a wall is simple – a wall is hit when `headX` or `headY` have values of 0 or 30 (`$1E`) and 24 (`$18`) respectively. If walls are on (`wallsreg=$FF`), then a game over is triggered (section 15). If walls are off (`wallsreg=$00`), then `headX` and `headY` are set to the opposite position on the game grid (i.e. the head warps to the other side).

11) Game logic – hitting the tail

Detection of hitting the tail is slightly more complicated than in the case of the wall. `headY` is compared with every snake segment's y coord (by counting `X` no. of squares down from `Y` in SRAM). If there is a match, `headX` is compared with the associated segment x coord. If there is a match here as well, then a game over is triggered.

To save some time, the first three squares are omitted, since the head cannot hit those squares at all.

12) Game logic – eating a food

A food is eaten when there is a match between headX / headY, and foodX / foodY. When this happens, a single beep is triggered, and the length of the snake is increased by one. A new food is generated (see section **3**), and checked if it lies inside the snake in a very similar way to section **11**. If it does, a new food is generated, and this loop continues until a food is generated in a free spot.

The score on the LCD is then updated. To save time, the screen is not cleared, but rather three backspace commands are issued. The new three-digit score can then be output as per section **7** without having to output the whole line.

13) Moving the snake

If the snake passes the game logic steps, the head coords are added onto the SRAM record at Y, and Y is incremented by two (as each segment needs two bytes for x and y). At this point, the program checks if Y is approaching SRAM locations that are used for other purposes – specifically, the stack grows downwards from $0FFF, and avoiding it is essential. For this reason the program plays it safe and checks if YH, the Y high byte, is equal to $0E. If it is, the Ram_return routine is triggered.

A diagram of the operation of Ram_return is shown in fig. 5. The snake, in green, has reached the end of the allocated portion of SRAM. YH is equal to $0E, triggering the function. Z is loaded to $0102 + 2*X and the byte at Y gets copied to it. Y and Z are then both decremented by 1.

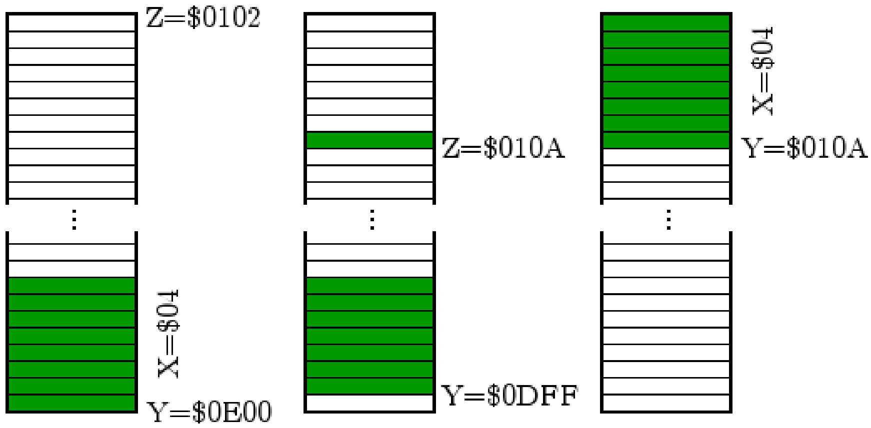


Fig. 5 Operation of Ram_return

This process is repeated 2*X number of times, and when finished Y is reset to the head of the snake again, $0102 + 2*X. In this way the entire snake is copied to the bottom of the allocated SRAM, ready to resume going upwards.

14) High scores

High scores are saved to the microcontroller's EEPROM. There are 6 possible high scores, corresponding to the combinations of difficulty and wall preference, and each high score is of word length (seeing as the score can – theoretically! – be above 256). Hence 12 bytes, at the very start of EEPROM, are used for keeping high scores. Gethsaddress retrieves the correct address for the high score based upon the two game options.

The high score menu allows to browse high scores for walls on and walls off. The three difficulties are shown on each page. To get each score, `TempReg` and `wallsreg` are changed to the appropriate values and `Gethsaddress` is called as if it were after a game.

Because every EEPROM byte initialises to `$FF` on a microcontroller erase option, the function `Clearhighscores`, accessible from the high score menu, must be run after every reprogram. It sets every high score to $000A – this is necessary because otherwise the program sees a high score of `$FFFF`, which is a 5 digit decimal number. This causes problems with `MessDecChar`.

15) Game over

When a game over is triggered, the function `Gameover` triggers the two-note descending game over beep, and checks if the current score is a high score, by retrieving the high score for the current game options via `Gethsaddress`. If the current score is higher, it is written into the same EEPROM address, overwriting the previous score.

16) Game over menu

`Gameovermenu` displays the current score and high score via `Decimaloutput`. If the player managed to achieve a high score in the last match, these two values will be equal. The player now can either replay the match with the same settings (jumping to section **3**) or go back to the main menu (jumping to section **2**).

Results and performance

The project satisfied everything set out in the original plan, including those features marked as extensions, e.g. high scores. The game works very well – however, there were a myriad of issues during development, some of which were eliminated, some minimised and some unfortunately could not be fixed.

Originally, the pixel display called for one databus and addressing via another port. This worked, but was quite slow, considering that the output of one pixel involved loading the x data, outputting it, loading the address data, outputting it, loading the y data, etc. By changing this to utilise two databuses, the output of one pixel was vastly speeded up. Seeing as this is a low-level operation, the gains were exponential.

It was also planned to have solid 10x10 squares on the game grid (a round 25x20 grid). It was soon discovered that solid squares were not feasible, as too many pixels were being drawn even for a fairly sparse screen, and screen flicker was unbearable. The 10x10 square idea was also quickly discarded when it was found that multiplication involved loading to another register and adding 10 times, whereas multiplication by 8 could be done much faster by only executing 3 logical shift rights. This also led to a large speed increase.
A minor issue that was fixed concerned the dot output on the oscilloscope screen. At the

end of a draw cycle, the electron beam would stay fixed on one place while the program ran through other instructions. This created an annoying bright spot on the screen. To solve this it was simply shifted to a position of screen in between draw cycles.

Another significant issue that was fixed was the LCD output. The `busylcd` function supplied contained a 49ms delay that ran every time. This led to very slow LCD performance as each character output had to suffer this delay. It was discovered that when outputting text, the delay could be omitted entirely and the display would still work perfectly (`busylcdshort`). This led to text appearing on the display nigh-instantly instead of visibly one by one, a huge improvement. However, other operations like clearing the display did not work without the delay. This was problematic – seeing as the oscilloscope screen output is synchronous, it would switch off whenever this delay was active. To solve this problem, a separate `busylcdsnake` function was successfully created, that instead of using a delay, drew the snake and walls.

As examples of issues that could not be fixed, the game noticeably slows down when the score becomes large. This is because the snake length affects several areas of the game: the drawing of the snake and the check for hitting the tail, and the probability that a food will have to be redrawn because it lies inside the snake increases. Unfortunately there was no easy fix to this. There is also an annoying screen flicker at the end of a sound beep, possibly due to the interrupt taking time to disable itself?

Updates, modifications and improvements

Several modifications could be made to improve the project. The screen slowing down could perhaps be fixed by changing the drawing routines to operate on another timer interrupt, similar to the sound effects but much slower. That way it would be guaranteed to trigger regularly. `Buttonpress` would also have to be changed if this were implemented, to avoid behaviour unexpected by the player.

An exceedingly fine-toothed examination of the code would probably turn up several places where registers were pushed and popped unnecessarily, as others were available. `xgrid` and `ygrid` come to mind, seeing as they are almost exclusively used only while actually drawing. Game logic could use these registers instead of other temporary ones, leading to speed improvements. Of course, if the screen were converted to run on interrupts, this would no longer be the case!

Sound could have been improved by using the in-built frequency generator. Unfortunately, the output was on port B, which was already in use. This could have been circumvented by using the onboard DtA, essentially swapping the roles of ports B and F. However, the required `sts` command takes longer than `out`, so there might be some synchronisation issues.

Improvements could include the facility to save the player's name along with their highscore, similar to the 3-letter names on arcade machines. Different shaped levels could have been added, or obstacles could have been dropped at regular intervals to add to difficulty. The snake head could have been changed to some other shape which indicated the direction of movement. In short, there are a multitude of features that could have been added, given the time.

Conclusion

The goal of the project was to make a fully functional snake game, with difficulty settings and sound effects. This was successful, as well as adding in the extensions put down in the project plan, namely the high score system and wall preference. A wide variety of techniques learnt in the lectures were utilised, as well as several features that had to be puzzled out by looking at technical and jargon-filled documentation!

Overall, I am satisfied with the outcome of the project. The work was tough as almost every concept was completely new to me, but also very rewarding, and I feel as though my lab partner and I have produced a highly polished product.

Product specification

- LCD display – 20 x 4 characters
- 4x4 keyboard input
- Game screen: oscilloscope display on XY mode, 248x200px
- Game grid: 8px squares, a 31x25 game grid
- 3 game difficulty options – easy, medium, hard
- 2 wall preference options – on, off
- 6 persistent high scores using EEPROM, up to the maximum score possible of 667.
- 3 different sound effects – extensible sound system
- Easy replay function on game over
- Internal high score reset function

Bibliography

[1]: B. Kristinsson (2011), *Ardrand: The Arduino as a Hardware Random-Number Generator,* available at http://arxiv.org/pdf/1212.3777.pdf
[2]: *Pseudo-Random Binary Sequence (PRBS) by a Linear-Feedback Shift Register (LFSR) with a (2^N-1) Period*, available at http://www.eecircle.com/applets/009/LFSR.html
[3]: adapted from M. Neil, *Digital to Analog Converters*, Microprocessors Course
[4]: *Blockade, The Arcade Video Game by Gremlin Ind, Inc.*, available at http://www.arcade-history.com/?n=blockade&page=detail&id=287
[5]: adapted from R. Ward, T. Molteno (2007), *Table of Linear Feedback Shift Registers*, available at http://www.eej.ulst.ac.uk/~ian/modules/EEE515/files/old_files/lfsr/lfsr_table.pdf

Appendix – Assembler Code

```
; **  ATmega128(L) Assembly Language File - IAR Assembler Syntax **
; Several routines including setups and LCD have been copied from lecture notes

.DEVICE ATmega128
.include "m128def.inc"
.def wallsreg     =  r24 ; wall, $00 for off, $FF for on
.def headY        =  r23 ; snake head y coord
.def headX        =  r22 ; snake head x coord
.def buttonstate  =  r21 ; button state
.def xgrid        =  r19 ; game grid x coord
.def ygrid        =  r18 ; game grid y coord
.def foodY        =  r25 ; food y coord
.def foodX        =  r20 ; food x coord
.def TempReg      =  r16 ; temporary values
.def TempReg2     =  r17 ;
.def freqout      =  r2  ; sets the value of the counter which controls the frequency of
                         ;   sounds played
.def freqdurL     =  r3  ; a word which controls how long a sound plays for
.def freqdurH     =  r4  ;
.def one          =  r5  ; used as the number one in the sound interrupt code as r0-15
                         ;   can't be directly compared
.def zero         =  r6  ; used as zero
.def notecount    =  r7  ; Used a countdown for the number of notes left to be played in
                         ;   a sound
.def soundselect  =  r8  ; Used to select which sound to play


;****************** INTERRUPTS - SET UP *****************************

jmp Init         ; jmp is 2 word instruction to set correct vector
nop              ; Vector Addresses are 2 words apart
reti             ; External 0 interrupt  Vector
nop
reti             ; External 1 interrupt  Vector
nop
reti             ; External 2 interrupt  Vector
nop
reti             ; External 3 interrupt  Vector
nop
reti             ; External 4 interrupt  Vector
nop
reti             ; External 5 interrupt  Vector
nop
reti             ; External 6 interrupt  Vector
nop
reti             ; External 7 interrupt  Vector
nop
reti             ; Timer 2 Compare Vector
nop
reti             ; Timer 2 Overflow Vector
nop
reti             ; Timer 1 Capture  Vector
nop
reti             ; Timer1 CompareA  Vector
nop
reti             ; Timer 1 CompareB  Vector
nop
reti             ; Timer 1 Overflow  Vector
jmp Timer0comp   ; Timer 0 Compare  Vector
nop
reti             ; Timer 0 Overflow interrupt  Vector
nop
reti             ; SPI  Vector
```

```
nop
reti            ; UART Receive  Vector
nop
reti            ; UDR Empty  Vector
nop
reti            ; UART Transmit  Vector
nop
reti            ; ADC Conversion Complete Vector
nop
reti            ; EEPROM Ready Vector
nop
reti            ; Analog Comparator  Vector

;****************** INTERRUPTS ******************************

; INPUTS: freqout (r2), freqdur (r4:r3), notecount (r7),
;   soundselect (r8), and one & zero (r5,r6).
; Every time the compare on timer 0 hits (TCNT0 = OCR0), this is
;   called. The current freqout is inverted (on/off), and one is
;   subtracted from the freqdur word. Once freqdur hits 0, if
;   notecount is 0 the timer interrupt is disabled, terminating sound
;   output. If notecount is 1, a new value is given to OCR0 depending
;   on the value of soundselect, changing the frequency of the note.
;   A new value is given to freqdur, and then the interrupts run
;   as described above.
; OUTPUTS: TIMSK (last run)

Timer0comp:
                in r1,SREG          ; save SREG
                com freqout         ; inverts freqout
                sts $0062, freqout  ; out PORTF,freqout
                sub freqdurL,one    ; subtract one from word
                sbc freqdurH,zero   ;
                cp freqdurH,zero
                brne retinterrupt   ; if still playing, skip to reti
                cp freqdurL,zero
                brne retinterrupt   ; if we get here, freqdur = 0
                cp notecount,zero
                breq endsound       ; if no more notes, disable timer interrupt
                cp soundselect,zero ; if soundselect = 0, go to setstartsound
                breq setstartsound
                brne setendsound    ; if it's not, go to setendsound

setstartsound:  ldi TempReg,$03
                mov freqdurH, TempReg
                ldi TempReg,$FE
                mov freqdurL, TempReg
                ldi TempReg,$08     ; value which counter TCNT0 hits compare at
                out OCR0,TempReg
                sub notecount,one   ; no more additional notes
                rjmp retinterrupt

setendsound:    ldi TempReg,$00
                mov freqdurH, TempReg
                ldi TempReg,$FE
                mov freqdurL, TempReg
                ldi TempReg,$1E
                out OCR0,TempReg
                sub notecount,one
                rjmp retinterrupt

endsound:       push TempReg
                ldi TempReg,$00     ; when sound is over, disable the timer interrupt
                out TIMSK,TempReg
                pop TempReg

retinterrupt:   out SREG,r1         ; restore SREG
                reti
```

```
;****************** INITIALISATION *****************************

Init:

    ; ******* Stack Pointer ****
    ldi r16, $0F
    out SPH,r16     ; Stack Pointer High Byte
    ldi r16, $FF    ; Stack Pointer Setup
    out SPL,r16     ; Stack Pointer Low Byte

    ; ******* RAMPZ Setup ****
    ldi  r16, $00   ; 1 = EPLM acts on upper 64K
    out RAMPZ, r16  ; 0 = EPLM acts on lower 64K

    ; ******* Sleep Mode And SRAM  ****
    ldi r16, $C0    ; Idle Mode - SE bit in MCUCR not set
    out MCUCR, r16  ; External SRAM Enable Wait State Enabled

    ; ******* Comparator Setup ****
    ldi r16,$80    ; Comparator Disabled, Input Capture Disabled
    out ACSR, r16  ; Comparator Settings

    ; ******* Port A Setup ****  LCD
    ldi r16, $FF    ; Address every bit
    out DDRA, r16   ; Port A Direction Register
    ldi r16, $FF    ; Init value
    out PORTA, r16  ; Port A value

    ; ******* Port B Setup ****  Data Bus ycoord
    ldi r16, $FF
    out DDRB , r16  ; Port B Direction Register
    ldi r16, $99    ; Init value
    out PORTB, r16  ; Port B value

    ; ******* Port C Setup ****  LCD
    ldi r16, $FF
    out DDRC, r16   ; Port C Direction Register
    ldi r16, $FF    ; Init value
    out PORTC, r16  ; Port C value

    ; ******* Port D Setup ****  Data Bus xcoord
    ldi r16, $FF
    out DDRD, r16   ; Port D Direction Register
    ldi r16, $FF    ; Init value
    out PORTD, r16  ; Port D value

    ; ******* Port E Setup ****  Button Input
    ; Set-up within button press routine.

    ; ******* Port F Setup ****  Sound
    ldi r16, $FF
    sts $0061, r16  ; Port F Direction Register
    ldi r16, $00    ; Init value
    sts $0062, r16  ; Port F value

    ; ******* LCD Setup ****  Display Initialization
    rcall DEL15ms         ; wait 15ms for things to relax after power up
    ldi TempReg,$30       ; Hitachi says do it...
    sts   $8000, TempReg  ; so i do it...
    rcall DEL4P1ms        ; Hitachi says wait 4.1 msec
    sts   $8000, TempReg  ; and again I do what I'm told
    rcall DEL100mus       ; wait 100 mus
    sts   $8000, TempReg  ; here we go again folks
    rcall busylcd
    ldi TempReg, $3F      ; Function Set : 2 lines + 5x7 Font
    sts  $8000, TempReg
    rcall busylcd
    ldi TempReg, $08      ; display off
    sts  $8000, TempReg
```

```
    rcall busylcd
    ldi TempReg, $01      ; display on
    sts  $8000, TempReg
    rcall busylcd
    ldi TempReg, $38      ; function set
    sts  $8000, TempReg
    rcall busylcd
    ldi TempReg, $0C      ; display on
    sts  $8000, TempReg
    rcall busylcd
    ldi TempReg, $06      ; entry mode set increment no shift
    sts  $8000, TempReg
    rcall busylcd
    clr TempReg

    ; ******* Interrupts Setup ****
    sei             ; enable all interrupts
    ldi r16, $00    ; timer interrupt enable (starts off, $02 enables)
    out TIMSK, r16  ; timer 0 compare

    ; ******* Timer0 Setup Code ****
    ldi r16,$0E     ; status register - prescaling, function set
    out TCCR0,r16   ;  (clear on compare)
    ldi r16,$0F     ; value which counter TCNT0 hits compare at
    out OCR0,r16

    ; ******* Timer Setup
    ldi r16,$01
    mov freqout,r16 ; doesn't really matter whether on/off
    ldi r16,$01
    mov one,r16
    ldi r16,$00
    mov zero,r16

;****************** MENUS ******************************

; INPUTS = none
; Intro menu. Can go to difficulty menu or highscore menu. TempReg2
;   is a timer constantly increasing, as this is a source of randomness.
; OUTPUTS = TempReg2 (r17) as timer

Intromenu:
            rcall CLRDIS       ; clear LCD
            rcall IntroOut     ; display intro message
            ldi TempReg2,$00
startloop:  rcall Splashscreen ; display introductory splash screen
            rcall Walldisplay  ; display walls
            rcall Buttonpress  ; detect button press
            inc TempReg2       ; increment the TempReg2 timer
            cpi TempReg,$E7
            breq Diffmenujmp   ; if button = A, go to difficulty
            cpi TempReg,$D7
            breq Hsmenu        ; if B, go to highscores
            rjmp startloop

Diffmenujmp:
            rjmp Diffmenu      ; branch out of reach

; INPUTS = none
; Highscore menu. Can display highscores for walls on/off,
;   go to the difficulty menu, or reset highscores.
; OUTPUTS = none

Hsmenu:
            rcall CLRDIS
            rcall HsIntroOut
hsloop:     rcall Splashscreen
            rcall Walldisplay
            rcall Buttonpress
```

```
            cpi TempReg,$EE
            breq Hswallondisp    ; if 1, walls on highscores
            cpi TempReg,$ED
            breq Hswalloffdisp   ; if 2, walls off highscores
            cpi TempReg,$E7
            breq Diffmenu        ; if A, difficulty
            cpi TempReg,$77
            breq Clearhsjmp      ; if D, clear high scores
            rjmp hsloop

Clearhsjmp:
            rjmp Clearhighscores

; INPUTS = none
; Displays highscores for walls on. Can only return to highscore menu.
; OUTPUTS = none

Hswallondisp:
            rcall CLRDIS
            rcall HswallondispOut
            ldi wallsreg,$FF        ; high scores read from current game variables, so we
set them here.
            ldi TempReg,$08         ; walls on, difficulty easy
            rcall Writehighscore
            rcall HseasyOut
            ldi TempReg,$12         ; difficulty medium
            rcall Writehighscore
            rcall HshardOut
            ldi TempReg,$04         ; difficulty hard
            rcall Writehighscore
            rcall HsretOut
hswonloop:  rcall Splashscreen
            rcall Walldisplay
            rcall Buttonpress
            cpi TempReg,$B7         ; if C, highscore menu
            breq Hsmenu
            rjmp hswonloop

; INPUTS = none
; Displays highscores for walls off. Can only return to highscore menu.
; OUTPUTS = none

Hswalloffdisp:
            rcall CLRDIS
            rcall HswalloffdispOut
            ldi wallsreg,$00        ; same as before, but walls off
            ldi TempReg,$08
            rcall Writehighscore
            rcall HseasyOut
            ldi TempReg,$12
            rcall Writehighscore
            rcall HshardOut
            ldi TempReg,$04
            rcall Writehighscore
            rcall HsretOut
hswoffloop: rcall Splashscreen
            rcall Walldisplay
            rcall Buttonpress
            cpi TempReg,$B7
            breq Hsmenu
            rjmp hswoffloop

; INPUTS = TempReg2 (r17)
; Difficulty menu. First stores the TempReg2 menu timer as a random seed,
;   then options allow to set difficulty.
; OUTPUTS = TempReg2 (r17) as timer, TempReg (r16) as difficulty,
;   food x seed in SRAM ($0100)

Diffmenu:
```

```
            sts $0100,TempReg2  ; uses time to press button on intro menu as random seed
for x
            rcall CLRDIS
            rcall DiffOut
            ldi TempReg2,$00
diffloop:   rcall Splashscreen
            rcall Walldisplay
            rcall Buttonpress
            inc TempReg2        ; using TempReg2 as a timer again
            cpi TempReg,$EE
            breq setdiffeasy    ; if 1, easy
            cpi TempReg,$ED
            breq setdiffmed     ; if 2, medium
            cpi TempReg,$EB
            breq setdiffhard    ; if 3, hard
            rjmp diffloop

setdiffeasy:
            ldi TempReg,$12     ; screen refreshes per game update
            rcall CLRDIS
            rcall SelectOut     ; confirmation message
            rcall DiffEasyOut   ;
            rcall SplashDEL     ; delay, but using splash and wall displays so no blank
screen
            rjmp Wallmenu

setdiffmed:
            ldi TempReg,$08
            rcall CLRDIS
            rcall SelectOut
            rcall DiffMedOut
            rcall SplashDEL
            rjmp Wallmenu

setdiffhard:
            ldi TempReg,$04
            rcall CLRDIS
            rcall SelectOut
            rcall DiffHardOut
            rcall SplashDEL
            rjmp Wallmenu

; INPUTS = TempReg2 (r17)
; Walls menu. First stores the TempReg2 menu timer as a random seed,
;   then options allow to set the wall preference.
; OUTPUTS = wallsreg (r24), food y seed in SRAM ($0101)

Wallmenu:
            push TempReg
            sts $0101,TempReg2  ; uses time to press button on diff menu as random seed
for y
            rcall CLRDIS
            rcall WallsOut
wallloop:   rcall Splashscreen
            rcall Walldisplay
            rcall Buttonpress
            cpi TempReg,$EE
            breq setwallson     ; if 1, walls on
            cpi TempReg,$ED
            breq setwallsoff    ; if 2, walls off
            rjmp wallloop

setwallson:
            ldi wallsreg,$FF
            rcall CLRDIS
            rcall SelectOut
            rcall WallsOnOut
            rcall SplashDEL
            rjmp Gamesetup
```

```
setwallsoff:
            ldi wallsreg,$00
            rcall CLRDIS
            rcall SelectOut
            rcall WallsOffOut
            rcall SplashDEL
            rjmp Gamesetup

; INPUTS = none
; General game initialisation routine. Sets everything up for a new game.
; OUTPUTS = foodX & foodY (r20,r25), headX & headY (r22,r23),
;   X as snake length, Y as head location in SRAM, buttonstate (r21)

Gamesetup:
    pop TempReg     ; pops the push from Wallmenu
    rcall CLRDIS
    rcall Randomfood
    ldi headX, $19  ; head location (top left)
    ldi headY, $05
    ldi XH, $00     ; initial snake length
    ldi XL, $0A
    ldi YH, $01     ; initial SRAM location
    ldi YL, $02
    st Y+,headX     ; stores initial snake segments in SRAM
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y+,headY
    dec headX
    st Y+,headX
    st Y,headY              ; Y is last coord, don't increment here
    ldi buttonstate, $DB    ; initial direction (right)
    rcall Gamestartbeep
    rcall ScoreIntroOut     ; score message
    rcall Decimaloutput
    rjmp Gameplay

; INPUTS = none
; Game over menu. Displays the game over splash screen, the score,
;   and the high score achieved with the current game settings.
;   Can either replay game with same settings or go to main menu.
; OUTPUTS = none

Gameovermenu:
            push TempReg         ; we need to keep TempReg for replays
            rcall CLRDIS
            rcall EndMenu1Out    ; display score message
```

```
            rcall Decimaloutput  ;
            rcall EndMenu2Out    ;
            rcall EndMenu3Out    ;
            rcall Writehighscore ; this is the highscore line
            rcall EndMenu4Out    ;
endmenuloop:inc TempReg2
            rcall Gameoverscreen
            rcall Walldisplay
            rcall Buttonpress
            cpi TempReg, $E7
            breq Gamesetup       ; if A, new game with same settings
            cpi TempReg, $B7
            breq NewGame         ; if C, main menu
            rjmp endmenuloop

NewGame:
            rjmp Intromenu      ; branch out of reach

;******************** SOUNDS ********************************************

; INPUTS = none
; Enables the compare interrupt which starts outputting sound. Also sets
;   an additional higher note for a game start.
; OUTPUTS = freqdur (r4:r3), TIMSK, notecount (r7), soundselect (r8)

Gamestartbeep:
    push TempReg
    ldi TempReg,$01
    mov freqdurH, TempReg
    ldi TempReg,$FF
    mov freqdurL, TempReg     ; loads $01FF into freqdur
    ldi TempReg,$02
    out TIMSK, TempReg        ; enable timer 0 compare interrupt
    ldi TempReg,$01
    mov notecount,TempReg     ; additional note!
    ldi TempReg,$00
    mov soundselect,TempReg   ; sets additional note type
    pop TempReg
    ret

; INPUTS = none
; Enables the compare interrupt which starts outputting sound. Also sets
;   an additional lower note for game over.
; OUTPUTS = freqdur (r4:r3), TIMSK, notecount (r7), soundselect (r8)

Gameendbeep:
    push TempReg
    ldi TempReg,$01
    mov freqdurH, TempReg
    ldi TempReg,$FF
    mov freqdurL, TempReg
    ldi TempReg,$02
    out TIMSK, TempReg
    ldi TempReg,$01
    mov notecount,TempReg
    ldi TempReg,$01
    mov soundselect,TempReg
    pop TempReg
    ret

; INPUTS = none
; Enables the compare interrupt which starts outputting sound.
; OUTPUTS = freqdur (r4:r3), TIMSK, notecount (r7), soundselect (r8)

Beep:
    push TempReg
    ldi TempReg,$01
    mov freqdurH, TempReg
    ldi TempReg,$FF
```

```
    mov freqdurL, TempReg
    ldi TempReg,$02
    out TIMSK, TempReg
    ldi TempReg,$00
    mov notecount,TempReg     ; no additional notes
    pop TempReg
    ret

;******************** HIGH SCORE ROUTINES *******************************

; INPUTS = r18 as data, r20:r19 as data address
; Writes r18 to the address r20:r19.
; OUTPUTS = none

EEPROM_write:
    sbic EECR,EEWE    ; wait for completion of previous write
    rjmp EEPROM_write
    out EEARH, r20    ; set up address in address register
    out EEARL, r19
    out EEDR,r18      ; set up data in data register
    sbi EECR,EEMWE    ; write logical one to EEMWE
    sbi EECR,EEWE     ; start eeprom write by setting EEWE
    ret

; INPUTS = r20:r19 as data address
; Reads from address r20:r19 and puts into r18.
; OUTPUTS = r18 as data

EEPROM_read:
      sbic EECR,EEWE    ; wait for completion of previous read
      rjmp EEPROM_read
      out EEARH, r20    ; set up address
      out EEARL, r19
      sbi EECR,EERE     ; start eeprom read by setting EERE
      in r18,EEDR       ; read data from data register
      ret

; INPUTS = none
; Goes through highscore area of EEPROM, setting every highscore
;   to $00A0 (initial snake length is 10, which doesn't count).
;   Necessary because every EEPROM byte initialises to $FF on a
;   program write (also for jealous/angry gamers).
; OUTPUTS = none

Clearhighscores:
            ldi r20,$00         ; set initial address to 0
            ldi r19,$00
clhs10loop: ldi r18,$0A         ; minimum score is 10 (snake length) for decoutput
            rcall EEPROM_write
            inc r19
            cpi r19,$0C
            brne clhs1loop      ; go to next loop
clhs1loop:  ldi r18,$00         ; this is because scores are words - have to set $000A
            rcall EEPROM_write
            inc r19
            cpi r19,$0C
            brne clhs10loop
clhsret:    rcall CLRDIS
            rcall HsresetOut
            rcall SplashDEL
            rjmp Hsmenu         ; go to previous loop

; INPUTS = wallsreg (r24), TempReg (r16) as difficulty
; Given the two game parameters, outputs the address
;   for the corresponding highscore in EEPROM.
;   2 bytes per address: won:easy,med,hard; woff:easy,med,hard.
; OUTPUTS = r20:r19 as EEPROM data address

Gethsaddress:
```

```
            ldi r20,$00
            ldi r19,$00
            cpi wallsreg,$FF
            breq gethsdiff  ; if walls on, go to difficulty
            subi r19,$FA    ; if walls off, add 6 to address
gethsdiff:  cpi TempReg,$12
            breq gethsret
            subi r19,$FE    ; add 2 to address
            cpi TempReg,$08
            breq gethsret
            subi r19,$FE    ; add 2 to address
gethsret:   ret

; INPUTS = wallsreg (r24), TempReg (r16) as difficulty
; Gets the highscore from EEPROM and puts it into Y (highscores are words).
; OUTPUTS = Y as highscore

Gethighscore:
    rcall Gethsaddress
    rcall EEPROM_read
    mov YL,r18
    inc r19
    rcall EEPROM_read
    mov YH,r18
    ret

; INPUTS = X, as current game score
; Checks to see if the current game score is a highscore or not.
;   If it is, calls Sethighscore. Else, nothing.
; OUTPUTS = none

Checkhighscore:
            rcall Gethighscore
            cp YH,XH    ; if YH < XH, new highscore
            brlt Sethighscore
            brne checkhsret  ; if not equal, must be less than - no hs
            cp YL,XL    ; if YL < XL (and YH = XH), new hs
            brlt Sethighscore
checkhsret: ret

; INPUTS = X, as current game score
; Writes the current game score into the EEPROM address
;   corresponding to the game settings.
; OUTPUTS = none

Sethighscore:
    rcall Gethsaddress
    mov r18,XL
    rcall EEPROM_write
    inc r19
    mov r18,XH
    rcall EEPROM_write
    ret

;******************** MAIN GAME LOOP ************************************

; INPUTS = TempReg (r16) as difficulty
; The main gameplay loop. Draws the snake, food, and the walls TempReg
;   times (the more, the easier the game), then triggers a game logic step.
; OUTPUTS = none

Gameplay:
            push TempReg
gameloop:   rcall Drawsnake         ; draws the snake and food
            rcall Walldisplay       ; draws the walls
            dec TempReg
            cpi TempReg,$00         ; repeat TempReg times
            brne gameloop           ; when TempReg = 0, trigger game logic
            pop TempReg
```

```
            rcall Directionpress    ; get button input for new direction
            rcall Movehead          ; move the head according to current direction
            rcall Hitwall           ; the 3 logic steps
            rcall Tailcheck         ;
            rcall Eatfood           ;
            rcall Movesnake         ; if no Gameover triggered, move the snake.
            rjmp Gameplay

;******************** GAME LOGIC ************************************

; INPUTS = none
; If not rcalled from a function, pops to fix the stack. Starts off the
;   gameover beep, then checks if the score is a highscore.
; OUTPUTS = none

Gameovertail:
            pop TempReg2            ; didn't return from a called function, so
            pop TempReg2            ;   dump location of ret in TempReg2
            pop XL
            pop XH
Gameover:   rcall Gameendbeep       ; starts game over beep
            rcall Checkhighscore    ; checks highscore
            rjmp Gameovermenu

;**** WALLS ****

; INPUTS = headX & headY (r22,r23), wallsreg (r24)
; Checks if the head is in a wall. If it is and walls are on,
;   trigger gameover. If it is and walls are off, warp to other
;   side of screen. Else, nothing.
; OUTPUTS = none

Hitwall:
    cpi headX,$00
    breq hitright   ; hit right wall
    cpi headX,$1E
    breq hitleft    ; hit left wall
    cpi headY,$00
    breq hitup      ; hit top wall
    cpi headY,$18
    breq hitdown    ; hit bottom wall
    ret

hitright:
    cpi wallsreg,$FF
    breq Gameover
    ldi headX,$1D
    ret

hitleft:
    cpi wallsreg,$FF
    breq Gameover
    ldi headX,$01
    ret

hitup:
    cpi wallsreg,$FF
    breq Gameover
    ldi headY,$17
    ret

hitdown:
    cpi wallsreg,$FF
    breq Gameover
    ldi headY,$01
    ret

;**** TAIL ****
```

```
; INPUTS = headX & headY (r22,r23), X as snake length,
;   Y as head location in SRAM.
; Checks if the head has hit the tail. If it did,
;   trigger gameover. Else, nothing.
; OUTPUTS = none

Tailcheck:
        push YH
        push YL
        push TempReg2
        push XH
        push XL
        sbiw Y,$06       ; impossible to hit first 3 squares
        sbiw X,$04
        rcall checkYloop
        pop XL
        pop XH
        pop TempReg2
        pop YL
        pop YH
        ret

checkYloop:
        ld TempReg2,Y       ; loads Y coord into TempReg2
        sbiw Y,$02          ; goes to next Y, 2 down
        cp TempReg2,headY   ; if headY = Y coord, check X
        breq checkXloop
notx:   sbiw X,$01          ; count down snake length
        cpi XL,$00
        brne checkYloop
        cpi XH,$00
        brne checkYloop     ; loop for length of snake
        ret

checkXloop:
        adiw Y,$01          ; count up for the corresponding X coord
        ld TempReg2,Y       ; load into TempReg2
        sbiw Y,$01          ; go back to the Y coord SRAM position
        cp TempReg2,headX
        breq Gameovertail   ; if headX = X coord, game over
        brne notx           ; else keep checking other X

;**** FOOD ****

; INPUTS = headX & headY (r22,r23), foodX & foodY (r20,r25), X as snake length
; Checks if the head has hit a food. If it did, move the snake, update the score
;   on the LCD and generate a new food, making sure it's not in the tail.
; OUTPUTS = none

Eatfood:
            cpse headX,foodX
            ret
            cpse headY,foodY
            ret
            rcall Beep          ; start off the score beep
            adiw X,$01
            rcall Movesnake     ; Scoreupdate draws snake while running, so Movesnake
            rcall Scoreupdate   ;   must be called first to avoid tail flicker
newfoodloop:rcall Randomfood    ; places new food at random
            rjmp Foodtailcheck  ; checks new food isn't in tail

; INPUTS = foodX & foodY (r20,r25), X as snake length, Y as head location in SRAM
; Checks if the new food lies inside the tail of the snake. If it does,
;   generates a new food until it doesn't.
; OUTPUTS = none

Foodtailcheck:
                push YH
                push YL
```

```
                push TempReg2
                push XH
                push XL
foodcheckYloop: ld TempReg2,Y       ; loads Y coord into TempReg2
                sbiw Y,$02          ; goes to next Y, 2 down
                cp TempReg2,foodY   ; if foodY = Y coord, check X
                breq foodcheckXloop
foodnotx:       sbiw X,$01          ; count down snake length
                cpi XL,$00          ; loop for length of snake
                brne foodcheckYloop
                cpi XH,$00
                brne foodcheckYloop
                rjmp newfoodset     ; if code reaches this, new food has been set that's
                                    ;   not in the snake
foodcheckXloop: adiw Y,$01          ; count up for the X coord
                ld TempReg2,Y       ; load into TempReg2
                sbiw Y,$01          ; go back to initial SRAM position
                cp TempReg2,foodX
                breq foodintail     ; if headX = X coord, food is in tail
                brne foodnotx       ; else keep checking Y

foodintail:     pop XL
                pop XH
                pop TempReg2
                pop YL
                pop YH
                rjmp newfoodloop    ; food is in tail so set another food

newfoodset:     pop XL
                pop XH
                pop TempReg2
                pop YL
                pop YH
                rjmp Gameplay       ; jump back to start of gameplay loop as Movesnake
                                    ;   has already been done


; INPUTS = none
; 3 backspaces on the LCD, and outputs the new decimal score to LCD.
; OUTPUTS = none

Scoreupdate:
    push TempReg
    ldi TempReg, $10        ; backspace
    sts  $8000, TempReg
    rcall busylcdsnake      ; busylcd, but drawing snake and walls for no flicker
    ldi TempReg, $10
    sts  $8000, TempReg
    rcall busylcdsnake
    ldi TempReg, $10
    sts  $8000, TempReg
    rcall busylcdsnake
    rcall Decimaloutput     ; new decimal score output
    pop TempReg
    ret

;******************** SNAKE MOVES ***********************************

; INPUTS = buttonstate (r21)
; Moves the head of the snake according to the direction in buttonpress.
; OUTPUTS = headX (r22) or headY (r23)

Movehead:
        cpi buttonstate, $ED
        breq goup
        cpi buttonstate, $BD
        breq godown
        cpi buttonstate, $DE
        breq goleft
```

```
        cpi buttonstate, $DB
        breq goright
        ret

goup:   dec headY
        ret

godown: inc headY
        ret

goleft: inc headX
        ret

goright:dec headX
        ret

; INPUTS = headX & headY (r22,r23)
; Adds the current head location to the top of the snake SRAM.
;   additionally, checks if this is encroaching on SRAM locations
;   that are used for outputting to LCD.
; OUTPUTS = Y as head location in SRAM

Movesnake:
    push TempReg
    adiw Y,$01          ; increase snake save position
    st Y,headX          ; save snake head x coord
    adiw Y,$01          ; increase snake save position
    st Y,headY          ; save snake head y coord
    ldi TempReg,$0E
    cpse YH,TempReg     ; see if end of snake part of ram has been reached
    rjmp No_ram_return
    rjmp Ram_return

; INPUTS = none
; If SRAM location is fine, return to main gameplay loop.
; OUTPUTS = none

No_ram_return:
    pop TempReg
    ret

; INPUTS = X as snake length, Y as head location in SRAM
; If SRAM location is too high, need to relocate the snake SRAM
;   position. Z is set to $0102 plus the SRAM snake length, and
;   then bytes are sequentially copied from the Y address to the
;   Z address. Finally, Y is reset to the new head location in SRAM.
; OUTPUTS = Y as head location in SRAM

Ram_return:
            push XH         ; push variables
            push XL
            add XL,XL       ; double snake length - 2 bytes per square
            adc XH,XH
            ldi ZH, $01     ; initial location in SRAM
            ldi ZL, $02
            add ZL, XL      ; moves location in SRAM by (length of snake * 2)
            adc ZH, XH
shiftsnake: ld TempReg,Y    ; loads snake coordinate into TempReg
            sbiw Y,$01      ; decrements SRAM position by 1
            st Z, TempReg   ; stores coord in new location
            sbiw Z,$01      ; decrements new SRAM location & length by 1
            sbiw X, $01
            cpi XL,$00      ; loop if end of snake not reached
            brne shiftsnake
            cpi XH,$00
            brne shiftsnake
            ldi YH, $01     ; load initial SRAM location again
            ldi YL, $02
            pop XL          ; pop variables
```

```
            pop XH
            add YL, XL      ; move head location by (length of snake * 2)
            adc YH, XH
            add YL, XL
            adc YH, XH
            pop TempReg
            ret

;******************** BUTTON DECODE ***********************************

; INPUTS = none
; Gets the byte value input from the 4x4 keyboard
;   and saves into TempReg
; OUTPUTS = TempReg (r16) as byte input

Buttonpress:
    ldi TempReg, $F0    ; column
    out DDRE, TempReg
    ldi TempReg, $0F
    out PORTE, TempReg
    rcall DEL100mus
    in TempReg2,PINE
    ldi TempReg, $0F    ; row
    out DDRE , TempReg
    ldi TempReg, $F0
    out PORTE, TempReg
    rcall DEL100mus
    in TempReg,PINE
    or TempReg,TempReg2 ; combine into byte
    ret

; INPUTS = none
; Calls Buttonpress and checks if it is a direction (2,4,6,8).
;   If it is, checks if that direction is valid (snake can't go
;   right if it is currently going left). If it's valid, saves
;   to buttonstate. Else, nothing.
; OUTPUTS = buttonstate (r21)

Directionpress:
            push TempReg
            push TempReg2
            rcall Buttonpress
            cpi TempReg,$FF
            breq endbutton          ; skips checks if no button
            cpi TempReg,$ED
            breq checkup            ; if 2, check up
            cpi TempReg,$BD
            breq checkdown          ; if 8, check down
            cpi TempReg,$DE
            breq checkleft          ; if 4, check left
            cpi TempReg,$DB
            breq checkright         ; if 6, check right
            rjmp endbutton
setbutton:  mov buttonstate,TempReg
endbutton:  pop TempReg2
            pop TempReg
            ret

checkup:    ldi TempReg2,$BD            ; checks for down
            cpse buttonstate,TempReg2
            rjmp setbutton
            rjmp endbutton

checkdown:  ldi TempReg2,$ED            ; checks for up
            cpse buttonstate,TempReg2
            rjmp setbutton
            rjmp endbutton

checkleft:  ldi TempReg2,$DB            ; checks for right
```

```
            cpse buttonstate,TempReg2
            rjmp setbutton
            rjmp endbutton

checkright: ldi TempReg2,$DE            ; checks for left
            cpse buttonstate,TempReg2
            rjmp setbutton
            rjmp endbutton

;******************* RANDOM NUMBERS *******************************************

; INPUTS = X & Y seeds in SRAM ($0100,$0101)
; Executes a linear feedback shift generator in Galois configuration.
; OUTPUTS = foodX & foodY (r20,r25)

Randomfood:
            push TempReg
            push TempReg2
            lds r31,$0100       ; loads X seed into r31
            rcall getrandom     ; gets random number from seed
            sts $0100,TempReg   ; this becomes the new seed
ranloopX:   cpi TempReg,$1D
            brlo setX
            subi TempReg,$1D
            rjmp ranloopX       ; modulo loop to get a value within the game grid
setX:       inc TempReg
            mov foodX,TempReg   ; once done, set foodX.

            lds r31,$0101       ; loads Y seed into r31
            rcall getrandom
            sts $0101,TempReg   ; same as above but for Y
ranloopY:   cpi TempReg,$17
            brlo setY
            subi TempReg,$17
            rjmp ranloopY
setY:       inc TempReg
            mov foodY,TempReg

            pop TempReg2
            pop TempReg
            ret

getrandom:
            sbrs r31,0          ; skip if least significant bit is 1
            rjmp srinput0       ; if least significant bit is 1 jump to srinput0
            rjmp xor            ; if not go to xor

xor:
            ldi TempReg, $B1    ; $B1 is 1st,5th,6th,8th bits - the tap configuration
            ldi TempReg2, $B1
            and TempReg,r31     ; get bits to be toggled from the tap configuration
(1,5,6,8)
            eor TempReg,TempReg2; do toggling, TempReg becomes toggled bits
            com TempReg2
            and r31,TempReg2    ; r31 is bits that weren't toggled
            or r31,TempReg      ; r31 is now new random number
            lsr r31             ; shift byte to right
            sbr r31,$80         ; set highest bit
            rjmp retrandom

srinput0:
            lsr r31             ; shift byte to right
            cbr r31,$80         ; clear highest bit
            rjmp retrandom

retrandom:  mov TempReg,r31     ; TempReg becomes the new random number
            ret

;******************* SCOPE DISPLAY ROUTINES ************************************
```

```
; INPUTS = X as snake length, Y as head location in SRAM, foodX & foodY (r20,r25)
; Loads the Y address into ygrid, decrements Y by 1, and loads into
;   xgrid. Then calls Squaredisplay. Repeats this for length of snake.
;   Additionally, draws the food.
; OUTPUTS = xgrid & ygrid (r19,r18)

Drawsnake:
        push XH
        push XL
        push YH
        push YL
nextseg:ld ygrid, Y         ; loads head Y of snake into ygrid
        sbiw Y,$01
        ld xgrid, Y         ; loads head X of snake into xgrid
        sbiw Y,$01
        rcall Squaredisplay ; draws square
        sbiw X, $01
        cpi XH,$00
        brne nextseg
        cpi XL,$00
        brne nextseg
        mov xgrid,foodX
        mov ygrid,foodY
        rcall Squaredisplay
        pop YL
        pop YH
        pop XL
        pop XH
        ret

splashdata:
  .db $02,$02,$03,$02,$04,$02,$05,$02,$07,$02,$0B,$02,$0D,$02,$0E,$02,$0F,
$02,$10,$02,$11,$02,$13,$02,$17,$02,$19,$02,$1A,$02,$1B,$02,$1C,$02,$05,$03,$08,$03,$0B,
$03,$0D,$03,$11,$03,$13,$03,$16,$03,$17,$03,$1C,$03,$05,$04,$09,$04,$0B,$04,$0D,
$04,$11,$04,$13,$04,$15,$04,$17,$04,$1C,$04,$02,$05,$03,$05,$04,$05,$05,$05,$0A,$05,$0B,
$05,$0D,$05,$0E,$05,$0F,$05,$10,$05,$11,$05,$13,$05,$14,$05,$17,$05,$19,$05,$1A,$05,$1B,
$05,$1C,$05,$05,$06,$09,$06,$0B,$06,$0D,
$06,$11,$06,$13,$06,$17,$06,$19,$06,$05,$07,$08,$07,$0B,$07,$0D,
$07,$11,$07,$13,$07,$17,$07,$19,$07,$02,$08,$03,$08,$04,$08,$05,$08,$07,$08,$0B,$08,$0D,
$08,$11,$08,$13,$08,$17,$08,$19,$08,$1A,$08,$1B,$08,$1C,$08,$02,$0A,$03,$0A,$04,$0A,
$05,$0A,$06,$0A,$07,$0A,$08,$0A,$09,$0A,$0A,$0A,$0B,$0A,$0C,$0A,$0D,$0A,$0E,$0A,$0F,$0A,
$10,$0A,$11,$0A,$12,$0A,$13,$0A,$14,$0A,$15,$0A,$16,$0A,$17,$0A,$18,$0A,$1B,$0A,$02,$0B,
$02,$0C,$08,$0C,$09,$0C,$0A,$0C,$0C,$0C,$0E,$0C,$12,$0C,$14,$0C,$15,$0C,$16,$0C,$17,$0C,
$1A,$0C,$1B,$0C,$1C,$0C,$02,$0D,$07,$0D,$0A,$0D,$0C,$0D,$0E,$0D,$12,$0D,$14,$0D,$17,$0D,
$19,$0D,$1C,$0D,$02,$0E,$04,$0E,$07,$0E,$0A,$0E,$0C,$0E,$0E,$0E,$12,$0E,$14,$0E,$15,$0E,
$16,$0E,$17,$0E,$19,$0E,$1C,$0E,$02,$0F,$04,$0F,$07,$0F,$0A,$0F,$0C,$0F,$0F,$0F,$11,$0F,
$14,$0F,$17,$0F,$19,$0F,$1C,$0F,$02,$10,$04,$10,$08,$10,$09,$10,$0A,$10,$0C,
$10,$10,$10,$14,$10,$17,$10,$1A,$10,$1B,$10,$1C,
$10,$02,$11,$04,$11,$02,$12,$04,$12,$07,$12,$0B,$12,$0D,$12,$0E,$12,$0F,
$12,$10,$12,$13,$12,$14,$12,$15,$12,$17,$12,$19,$12,$1A,$12,$1B,$12,$1C,
$12,$02,$13,$04,$13,$07,$13,$0A,$13,$0B,$13,$0D,
$13,$10,$13,$12,$13,$15,$13,$17,$13,$19,$13,$1C,$13,$02,$14,$04,$14,$07,$14,$09,$14,$0B,
$14,$0D,$14,$0E,$14,$0F,$14,$10,$14,$12,$14,$15,$14,$17,$14,$19,$14,$1A,$14,$1B,$14,$1C,
$14,$02,$15,$04,$15,$07,$15,$08,$15,$0B,$15,$0D,
$15,$10,$15,$12,$15,$15,$15,$17,$15,$19,$15,$1C,$15,$02,$16,$03,$16,$04,$16,$07,$16,$0B,
$16,$0D,$16,$10,$16,$13,$16,$14,$16,$15,$16,$17,$16,$19,$16,$1C,$16

; INPUTS = splashdata byte table
; Sequentially displays splashdata bytes by copying
;   to xgrid and ygrid.
; OUTPUTS = xgrid & ygrid (r19,r18)

Splashscreen:
            push TempReg
            ldi TempReg,$00
            ldi ZH, HIGH(2*splashdata)
            ldi ZL, LOW(2*splashdata)
splashloop: lpm xgrid,Z+
```

```
            lpm ygrid,Z+
            rcall Squaredisplay
            inc TempReg
            cpi TempReg,$EF         ; no. of splash grids
            brne splashloop
            pop TempReg
            ret

gameoverdata:
  .db $02,$02,$03,$02,$04,$02,$05,$02,$06,$02,$07,$02,$09,$02,$0E,
$02,$10,$02,$11,$02,$12,$02,$13,$02,$14,$02,$15,$02,$17,$02,$18,$02,$19,$02,$1A,$02,$1B,
$02,$1C,$02,$07,$03,$09,$03,$0A,$03,$0D,$03,$0E,$03,$10,$03,$15,$03,$1C,
$03,$07,$04,$09,$04,$0B,$04,$0C,$04,$0E,$04,$10,$04,$15,$04,$1C,$04,$07,$05,$09,$05,$0E,
$05,$10,$05,$15,$05,$1C,$05,$03,$06,$04,$06,$05,$06,$06,$06,$07,$06,$09,$06,$0E,
$06,$10,$06,$11,$06,$12,$06,$13,$06,$14,$06,$15,$06,$17,$06,$18,$06,$19,$06,$1A,$06,$1C,
$06,$07,$07,$09,$07,$0E,$07,$10,$07,$15,$07,$17,$07,$1C,$07,$07,$08,$09,$08,$0E,
$08,$10,$08,$15,$08,$17,$08,$1C,$08,$07,$09,$09,$09,$0E,$09,$10,$09,$15,$09,$17,$09,$1C,
$09,$07,$0A,$09,$0A,$0E,$0A,$10,$0A,$15,$0A,$17,$0A,$1C,$0A,$02,$0B,$03,$0B,$04,$0B,
$05,$0B,$06,$0B,$07,$0B,$09,$0B,$0E,$0B,$10,$0B,$15,$0B,$17,$0B,$18,$0B,$19,$0B,$1A,$0B,
$1B,$0B,$1C,$0B,$02,$0D,$03,$0D,$04,$0D,$05,$0D,$06,$0D,$07,$0D,$09,$0D,$0A,$0D,$0B,$0D,
$0C,$0D,$0D,$0D,$0E,$0D,$10,$0D,$15,$0D,$17,$0D,$18,$0D,$19,$0D,$1A,$0D,$1B,$0D,$1C,$0D,
$02,$0E,$07,$0E,$0E,$0E,$10,$0E,$15,$0E,$17,$0E,$1C,$0E,$02,$0F,$07,$0F,$0E,$0F,$10,$0F,
$15,$0F,$17,$0F,$1C,$0F,$02,$10,$07,$10,$0E,$10,$10,$10,$15,$10,$17,$10,$1C,
$10,$02,$11,$03,$11,$04,$11,$05,$11,$06,$11,$07,$11,$0A,$11,$0B,$11,$0C,$11,$0D,$11,$0E,
$11,$10,$11,$15,$11,$17,$11,$1C,$11,$06,$12,$07,$12,$0E,$12,$10,$12,$15,$12,$17,$12,$1C,
$12,$05,$13,$07,$13,$0E,$13,$10,$13,$15,$13,$17,$13,$1C,$13,$04,$14,$07,$14,$0E,
$14,$10,$14,$15,$14,$17,$14,$1C,$14,$03,$15,$07,$15,$0E,$15,$11,$15,$14,$15,$17,$15,$1C,
$15,$02,$16,$07,$16,$09,$16,$0A,$16,$0B,$16,$0C,$16,$0D,$16,$0E,
$16,$12,$16,$13,$16,$17,$16,$18,$16,$19,$16,$1A,$16,$1B,$16,$1C,$16

; INPUTS = gameoverdata byte table
; Sequentially displays gameoverdata bytes by copying
;   to xgrid and ygrid.
; OUTPUTS = xgrid & ygrid (r19,r18)

Gameoverscreen:
                push TempReg
                ldi TempReg,$00
                ldi ZH, HIGH(2*gameoverdata)
                ldi ZL, LOW(2*gameoverdata)
gameoverloop:   lpm xgrid,Z+
                lpm ygrid,Z+
                rcall Squaredisplay
                inc TempReg
                cpi TempReg,$CC     ; no. of splash grids
                brne gameoverloop
                pop TempReg
                ret

; INPUTS = none
; Draws the 4 lines of squares that comprise the walls of the game area.
; OUTPUTS = xgrid & ygrid (r19,r18)

Walldisplay:
            ldi xgrid,$00
            ldi ygrid,$00
downwloop:  rcall Squaredisplay
            inc ygrid
            cpi ygrid,$18
            brne downwloop
leftwloop:  rcall Squaredisplay
            inc xgrid
            cpi xgrid,$1E
            brne leftwloop
upwloop:    rcall Squaredisplay
            dec ygrid
            cpi ygrid,$00
            brne upwloop
rightwloop: rcall Squaredisplay
```

```
            dec xgrid
            cpi xgrid,$00
            brne rightwloop
            push r24            ; oscilloscope dot ends
            push r25            ;   off screen to avoid
            ldi r24,$FF         ;   flashing
            ldi r25,$FF         ;
            rcall Outapixel     ;
            pop r25             ;
            pop r24             ;
            ret

; INPUTS = xgrid & ygrid (r19,r18)
; Draws a square on the oscilloscope screen based on xgrid and ygrid.
; OUTPUTS = xcoord & ycoord (r25,r24)

Squaredisplay:
            .def xcoord      =  r25; pixel xcoord
            .def ycoord      =  r24; pixel ycoord
            .def sqcounter   =  r20; counter for squares output
            push r24
            push r25
            push r20
            push TempReg
            mov xcoord,xgrid
            mov ycoord,ygrid
            lsl xcoord          ; multiply by 8 (squares are 8 pixels wide)
            lsl xcoord          ;
            lsl xcoord          ;
            lsl ycoord          ;
            lsl ycoord          ;
            lsl ycoord          ;

            ldi TempReg,$00
squareloop: ldi sqcounter,$00   ; draws round square (down,left,up,right)
downloop:   rcall Outapixel
            inc ycoord
            inc ycoord
            inc sqcounter
            cpi sqcounter,$04
            brne downloop
            ldi sqcounter,$00
leftloop:   rcall Outapixel
            inc xcoord
            inc xcoord
            inc sqcounter
            cpi sqcounter,$04
            brne leftloop
            ldi sqcounter,$00
uploop:     rcall Outapixel
            dec ycoord
            dec ycoord
            inc sqcounter
            cpi sqcounter,$04
            brne uploop
            ldi sqcounter,$00
rightloop:  rcall Outapixel
            dec xcoord
            dec xcoord
            inc sqcounter
            cpi sqcounter,$04
            brne rightloop
            inc TempReg
            cpi TempReg,$02    ; repeat square for brightness
            brne squareloop
            pop TempReg
            pop r20
            pop r25
            pop r24
```

```
            ret

; INPUTS = xcoord & ycoord (r25,r24)
; Draws a pixel on the oscilloscope based on xcoord and ycoord.
; OUTPUTS = PORTD as x databus, PORTB as y databus

Outapixel:
    out PORTD, xcoord  ; put the x coord on the databus
    out PORTB, ycoord  ; put the y coord on the databus
    ret

;******************** LCD DISPLAY ROUTINES **************************************

IntroMsg:
.db "  Welcome to Snake    Press A for Menu   By David and Aidan   B for Highscores",0

; INPUTS = message data byte table
; Loads message data to Z. Same for every single message!
; OUTPUTS = Z as memory address of message data

IntroOut:
    ldi ZH, HIGH(2*IntroMsg)
    ldi ZL, LOW(2*IntroMsg)
    rjmp MessMore

HsIntroMsg:
.db "  Snake Highscores    A for game menu   1,2 for walls on/offD to reset highscore",0

HsIntroOut:
    ldi ZH, HIGH(2*HsIntroMsg)
    ldi ZL, LOW(2*HsIntroMsg)
    rjmp MessMore

HswallondispMsg:
.db "Walls on highscores Medium: ",0

HswallondispOut:
    ldi ZH, HIGH(2*HswallondispMsg)
    ldi ZL, LOW(2*HswallondispMsg)
    rjmp MessMore

HswalloffdispMsg:
.db "Walls off highscoresMedium: ",0

HswalloffdispOut:
    ldi ZH, HIGH(2*HswalloffdispMsg)
    ldi ZL, LOW(2*HswalloffdispMsg)
    rjmp MessMore

HseasyMsg:
.db "         Easy:   ",0

HseasyOut:
    ldi ZH, HIGH(2*HseasyMsg)
    ldi ZL, LOW(2*HseasyMsg)
    rjmp MessMore

HshardMsg:
.db "         Hard:   ",0

HshardOut:
    ldi ZH, HIGH(2*HshardMsg)
    ldi ZL, LOW(2*HshardMsg)
    rjmp MessMore

HsretMsg:
.db " C to ret",0

HsretOut:
```

```
    ldi ZH, HIGH(2*HsretMsg)
    ldi ZL, LOW(2*HsretMsg)
    rjmp MessMore

; I hate having to do this but it is really silly
;   there is no 'skip line' command for the LCD.
HsresetMsg:
.db "                                          Highscores reset",0

HsresetOut:
    ldi ZH, HIGH(2*HsresetMsg)
    ldi ZL, LOW(2*HsresetMsg)
    rjmp MessMore

DiffMsg:
.db "Enter difficulty:     2 for medium        1 for easy          3 for hard!",0

DiffOut:
    ldi ZH, HIGH(2*DiffMsg)
    ldi ZL, LOW(2*DiffMsg)
    rjmp MessMore

; more silliness
SelectMsg:
.db "                                         You picked ",0

SelectOut:
    ldi ZH, HIGH(2*SelectMsg)
    ldi ZL, LOW(2*SelectMsg)
    rjmp MessMore

DiffEasyMsg:
.db "easy",0

DiffEasyOut:
    ldi ZH, HIGH(2*DiffEasyMsg)
    ldi ZL, LOW(2*DiffEasyMsg)
    rjmp MessMore

DiffMedMsg:
.db "medium",0

DiffMedOut:
    ldi ZH, HIGH(2*DiffMedMsg)
    ldi ZL, LOW(2*DiffMedMsg)
    rjmp MessMore

DiffHardMsg:
.db "hard",0

DiffHardOut:
    ldi ZH, HIGH(2*DiffHardMsg)
    ldi ZL, LOW(2*DiffHardMsg)
    rjmp MessMore

WallsMsg:
.db "Solid walls?          2 for no            1 for yes",0

WallsOut:
    ldi ZH, HIGH(2*WallsMsg)
    ldi ZL, LOW(2*WallsMsg)
    rjmp MessMore

WallsOnMsg:
.db "on",0

WallsOnOut:
    ldi ZH, HIGH(2*WallsOnMsg)
    ldi ZL, LOW(2*WallsOnMsg)
```

```
    rjmp MessMore

WallsOffMsg:
.db "off",0

WallsOffOut:
    ldi ZH, HIGH(2*WallsOffMsg)
    ldi ZL, LOW(2*WallsOffMsg)
    rjmp MessMore

; again with the silliness.
ScoreIntroMsg:
.db "                                             Score: ",0

ScoreIntroOut:
    ldi ZH, HIGH(2*ScoreIntroMsg)
    ldi ZL, LOW(2*ScoreIntroMsg)
    rjmp MessMore

EndMenuMsg1:
.db "Your score was: ",0

EndMenu1Out:
    ldi ZH, HIGH(2*EndMenuMsg1)
    ldi ZL, LOW(2*EndMenuMsg1)
    rjmp MessMore

EndMenuMsg2:
.db " Press A to replay.  ",0

EndMenu2Out:
    ldi ZH, HIGH(2*EndMenuMsg2)
    ldi ZL, LOW(2*EndMenuMsg2)
    rjmp MessMore

EndMenuMsg3:
.db "High score is:  ",0

EndMenu3Out:
    ldi ZH, HIGH(2*EndMenuMsg3)
    ldi ZL, LOW(2*EndMenuMsg3)
    rjmp MessMore

EndMenuMsg4:
.db " Press C for menu.",0

EndMenu4Out:
    ldi ZH, HIGH(2*EndMenuMsg4)
    ldi ZL, LOW(2*EndMenuMsg4)
    rjmp MessMore

; INPUTS = X as snake length
; Outputs three digits, corresponding to the decimal number
;   of the snake length word X, minus 10.
; OUTPUTS = r24 as digit counter

Decimaloutput:
            push r24
            push r25
            push XH
            push XL
            push TempReg
            ldi r24, 0          ; counter
            sbiw X,$0A          ; take 10 from X, initial length doesn't count
check100:   cpi XH, $00
            brne dec100
            cpi XL, $64         ; see if the score is more than 100
            brsh dec100         ; if it is add to the 100 counter and then reduce by 100
            rjmp out100
```

```
dec100:     inc r24             ; increase counter by  1 and reduce X by 100
            sbiw X,$32          ;
            sbiw X,$32          ;
            rjmp check100
out100:     rcall MessDecChar   ; write the 100s digit
            ldi r24, $00        ; Reset the counter
            rjmp check10
check10:    cpi XL, $0A         ; repeat above process for 10s
            brsh dec10
            rjmp out10
dec10:      inc r24
            sbiw X, $0A
            rjmp check10
out10:      rcall MessDecChar
            ldi r24, $00
            rjmp check1
check1:     cpi XL, $01         ; repeat above process for 1s
            brsh dec1
            rjmp out1
dec1:       inc r24
            sbiw X, $01
            rjmp check1
out1:       rcall MessDecChar
            pop TempReg
            pop XL
            pop XH
            pop r25
            pop r24
            ret

DecOutput:
.db "0123456789",0

; INPUTS = r24 as digit counter
; Selects the correct character to output to LCD based on r24 counter.
; OUTPUTS = Digit character to SRAM

MessDecChar:
            ldi ZH, HIGH(2*DecOutput)
            ldi ZL, LOW(2*DecOutput)
messdecloop:cpi r24,$00
            breq messdecout
            dec r24
            adiw Z,$01
            rjmp messdecloop
messdecout: lpm r25, Z
            sts $C000, r25
            rcall busylcdshort
            ret

; INPUTS = none
; Gets the highscore for current game settings, places it into X
;   and outputs the 3 characters to LCD.
; OUTPUTS = none

Writehighscore:
    rcall Gethighscore
    mov XH,YH
    mov XL,YL
    rcall Decimaloutput
    ret

MessMore:   push r25
            push TempReg
MessLoop:   lpm r25, Z+
            cpi r25, $00
            breq MessEnd
            sts $C000, r25
            rcall busylcdshort
```

```
            rjmp MessLoop
MessEnd:    pop TempReg
            pop r25
            ret

; INPUTS = none
; Clears the LCD display
; OUTPUTS = none

CLRDIS:
    push TempReg
    ldi TempReg, $01    ; clear display and send cursor
    sts $8000, TempReg  ;   to the most left position
    rcall busylcdshort
    pop TempReg
    ret

; INPUTS = none
; Clears the LCD display, while avoiding screen flicker.
; OUTPUTS = none

CLRDISsnake:
    push TempReg
    ldi TempReg, $01
    sts $8000, TempReg
    rcall busylcdsnake
    pop TempReg
    ret

; INPUTS = none
; Holds here until the LCD finishes instruction, with delay
; OUTPUTS = none

busylcd:
    rcall Del49ms
    lds TempReg, $8000  ;access
    sbrs TempReg, 7  ;check busy bit  7
    ret    ;return if clear
    rjmp busylcd

; INPUTS = none
; Holds here until the LCD finishes instruction, with no delay.
;   Only works for outputting text, but obviously much faster.
; OUTPUTS = none

busylcdshort:
    lds TempReg, $8000  ;access
    sbrs TempReg, 7  ;check busy bit  7
    ret    ;return if clear
    rjmp busylcdshort

; INPUTS = none
; Holds here until the LCD finishes instruction, with delay.
;   Delay is the display of snake and walls, to avoid flickering.
; OUTPUTS = none

busylcdsnake:
    lds TempReg, $8000  ;access
    sbrs TempReg, 7  ;check busy bit  7
    ret    ;return if clear
    rcall Drawsnake
    rcall Walldisplay
    rjmp busylcdsnake


;********************   DELAY ROUTINES *******************************************

; INPUTS = none
; A delay that constantly outputs splash screen to avoid flicker in menus.
```

```
; OUTPUTS = none

SplashDEL:
            push TempReg
            ldi TempReg,$30
spdelloop:  dec TempReg
            rcall Splashscreen
            rcall Walldisplay
            cpi TempReg,$00
            brne spdelloop
            pop TempReg
            ret

; The rest of this is other assorted delays mostly used for the LCD.

BigDEL:
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    rcall Del49ms
    ret

DEL15ms:
;
; This is a 15 msec delay routine. Each cycle costs
;   rcall  -> 3 CC
;   ret    -> 4 CC
;   2*LDI  -> 2 CC
;   SBIW   -> 2 CC * 19997
;   BRNE   -> 1/2 CC * 19997
;
        LDI XH, HIGH(19997)
        LDI XL, LOW (19997)
COUNT:  SBIW XL, 1
        BRNE COUNT
        RET

DEL4P1ms:
        LDI XH, HIGH(5464)
        LDI XL, LOW (5464)
COUNT1: SBIW XL, 1
        BRNE COUNT1
        RET

DEL100mus:
        push XH
        push XL
        LDI XH, HIGH(131)
        LDI XL, LOW (131)
COUNT2: SBIW XL, 1
        BRNE COUNT2
        pop XL
        pop XH
        RET

DEL49ms:
        push XH
        push XL
```

```
        LDI XH, HIGH(65535)
        LDI XL, LOW (65535)
COUNT3: SBIW XL, 1
        BRNE COUNT3
        pop XL
        pop XH
        RET

DEL600mus:
        LDI XH, HIGH(798)
        LDI XL, LOW (798)
COUNT4: SBIW XL, 1
        BRNE COUNT4
        RET
```