Snack&Snake alpha: A New Vision to Happy Meal Toys

Daniel Weronski Falcó
Blackett Laboratory, Imperial College London, SW7 2AZ, UK

***Abstract*—This report introduces the Snack&Snake alpha prototype, an innovative PIC18 microcontroller-based gaming console specifically designed for the 'Snake' game. Targeted towards the fast-food industry, particularly as a novel addition to children's meals, the console emphasizes cost-effectiveness and sustainability. The report details technical development, focusing on hardware design, software logic, power considerations, and performance metrics. It explores commercial applications, offering a sustainable, low-maintenance, and robust alternative to conventional entertainment devices in fast-food settings.**

I. Introduction

THIS report outlines the development of a PIC18 microcontroller-based console designed to run the game 'Snake', with a novel market approach aimed at fast-food industry giants like McDonald's or Burger King. Initially conceived as a potential inclusion in children's meals, the project's scope has evolved to consider cost-effectiveness and environmental impact. A model where the console is available for purchase at a discounted rate, with games being the interchangeable items included in such meals is now proposed as an alternative, with potential expansion in use-cases. This shift not only reduces costs but also aligns with sustainability goals and environmentally-focused brand image such companies may wish to portray.

Furthermore, an alternative approach is introduced: deploying these robust and low-maintenance consoles in fast-food restaurant chains for on-site entertainment. This strategy contrasts with the use of multifunctional, high-maintenance devices like iPads, offering a simpler, more durable and cost-effective solution. This report delves into the technical development of the console and explores these innovative commercial applications, reflecting a practical, industry-aligned perspective.

II. High Level Design

The core principles outlined in the initial proposal report of modularity, performance and robustness have been meticulously followed. Parts of the design require precise timing control, mainly GLCD communication, and as a consequence some delay subroutines were implemented. Admittedly, this is a 'waste' of CPU cycles, but given the ease of implementation and relatively minor performance impact (battery-wise), it was deemed acceptable. Barring this minor detail, the prototype has been designed to be as efficient as possible, in line with the core design principles. The design is also highly modular: the code is split into numerous files, each with a specific purpose; this makes it easier to debug, maintain, extend, and reuse the code in the future by multiple potential developers. A number of hardware drivers were implemented in their own files, e.g. GLCD and keypad drivers, with the functions within, designed to be as generic and self-contained as possible.

A myriad of bugs and issues were encountered during the development of the project; while acknowledging the possibility of some bugs having gone undetected, all known issues were resolved.

In Fig. 1, the high-level logic flow of the program is depicted, with multiple distinct phases within the program's execution being observed. In the following sections, brief descriptions of each of these phases will be provided.

A. Initialisation

The program starts by initialising the hardware and some software features. This includes the following:

1) Initialises 256-byte long memory locations for the `x` and `y` components of the snake body
2) Initialises two timers from which a pseudo-random location for the food will be obtained every time it is needed
3) Initialises the GLCD and clears its contents should there be any
4) Initialises the keypad
5) Initialises the positions of the snake (on the top left of the screen)
6) Spawns the food.

B. User Input

After initialisation, the program enters a loop in which it waits for user input. Due to the way the keypad driver has been implemented, only four keys are valid (corresponding to right, left, up, and down snake movements) so the driver loops until one of these keys is pressed.

C. Snake Propagation Part I

Once a valid key is pressed, the program checks whether the current (yet to be updated) position of the snake head is on the border of the screen. If it is, the game will not allow the user to propagate into the wall triggering game termination, i.e. resets to initialisation. If the snake does not propagate into the walls, the position of the head of the snake (`x_pos`, `y_pos`) will be incremented or decremented in the `x` or `y` direction accordingly. After the new `x`, `y` positions have been calculated, the program checks whether the snake has crashed into itself, and terminates the game accordingly.

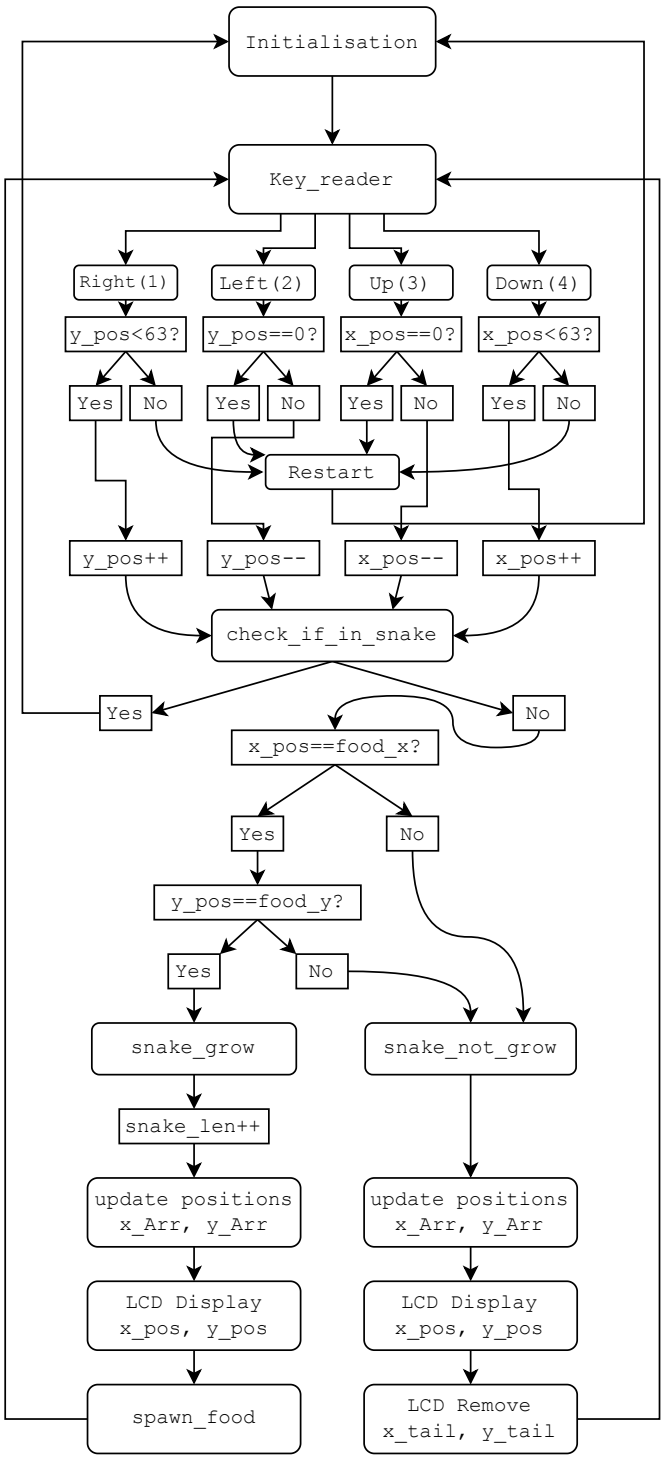

Fig. 1. High-level flowchart illustrating the primary logic flow from initialisaition to game progression, including user input handling, snake movement and growth mechanics, collision detection and the display update process.

D. Growth Check

After the new position of the snake head is computed, the program checks whether this position is the same as the food position. Whether the snake grows or not will determine the logical path the program takes in the following step.

E. Snake Propagation Part II

After the snake has grown (or not), the program will propagate the snake by one unit. This is done by turning on the pixel corresponding to `x_Arr[0]` and `y_Arr[0]` (equal to `x_pos` and `y_pos`) and turning off the pixel corresponding to `x_Arr[snake_len]` and `y_Arr[snake_len]` (which corresponds to the tail position) if the snake does not grow.

F. Array Update

After the new pixel is lit up, the program will update the `x_Arr` and `y_Arr` arrays. If the snake has grown, the program will shift all the elements of the array by incrementing the indeces by 1, the new position of the snake head is stored in `x_Arr[0]` and `y_Arr[0]`, and finally `len_snake` is incremented by 1.

The same occurs if the snake does not grow, except that the program does not increment `snake_len`, meaning that the last element of the array is ignored.

Finally, the program spawns a new food in a random location if the snake has grown.

III. Software Design

A high-level flowchart illustrating the primary logic flow can be seen from Fig. 1. However, there are a number of nuances not covered by this diagram; this section outlines the most important algorithmic components.

A. Initialisation

The program starts in the `main.s` file, which calls a number of routines external to it. An overview of the initialisation process can be seen in Fig. 2. It is important to note, that `x` and `y` are not in the traditional orientation, but rather `y` increases horizontally to the right and `x` increases vertically downward.

B. *check_if_in_snake*

This subroutine, located in `array.s`, checks whether the newly propagated position of the head is in the snake body. It does so by comparing `x_pos` with all the elements of `x_Arr` and for each of the matches of `x` it compares `y_Arr` in the same index with `y_pos`. If both coordinated match, the subroutine runs the `RESET` command which makes the program run from the begining having cleared the stack pointer to avoid stack overflow or memory corruption issues. This avoids any potential errors or conflicts that could arise from residual data in the stack.

There is a variant of this subroutine, `check_if_in_snake_food`, which is located in `randgen.s` which checks if the randomised food spawn location is in the snake or not to prevent said case.

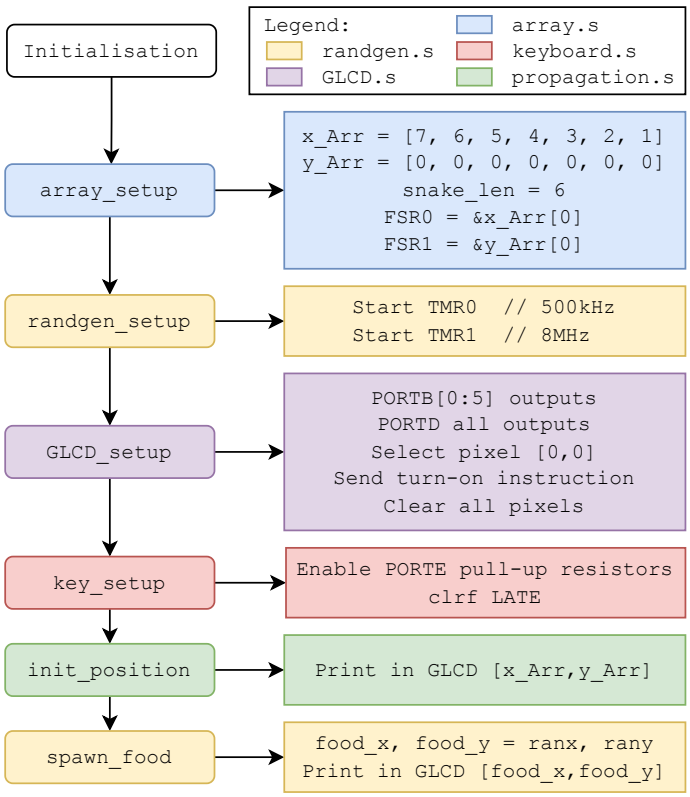

Fig. 2. Flowchart illustrating the initialisation process.

C. Update Position Arrays

This subroutine, located in `array.s`, is succinctly outlined in Fig. 3. It is called once the new position of the snake head has been computed, and has been checked to see if it is the same as the food position.

The arrays `x_Arr` and `y_Arr` are 256-byte long reserved memory spaces in which, initially, the first 7 elements are filled with the initial position of the snake; hence `snake_len=6` at first. The remaining elements after `Arr[6]` are simply ignored since all array-related routines work with FSR pointers and loop through the array until reaching `snake_len`.

The update routine works by copying all the elements of the array into the the next element (this is `Update 1` in Fig. 3). This is done by looping through the array elements, incrementing the index, and stopping at `snake_len` as mentioned previously. Once the copying is performed, the first element is overwritten with the current position (this is `Update 2` in Fig. 3). Finally, `snake_len` is either incremented in case the snake grows, or the routine simply returns otherwise.

D. Display Position (head/tail)

This routine, located in `propagation.s`, is responsible for turning on or off one particular pixel (head/tail) given `x` and `y` coordinates every time the snake propagates. Fig. 4 graphically outlines the algorithmic structure of this routine for the case of dealing with the head of the snake, `dsp_pos_head`.

The main challenge this routine faces is the fact that the GLCD is not a pixel-addressable device, but rather a byte-

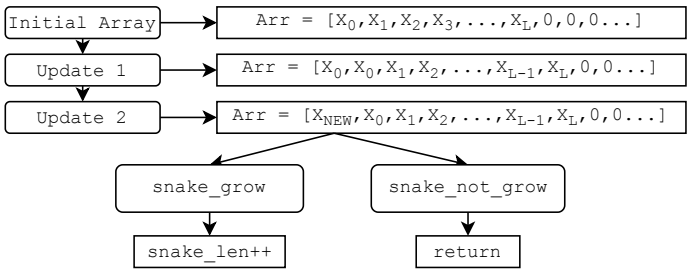

Fig. 3. Flowchart illustrating the update position arrays subroutine.

addressable one (i.e. it updates 8 pixels at a time). This means that the GLCD is divided into 8 8-bit wide rows (`x` direction) and 64 bit columns (`y` direction). As a consequence, while the `y` coordinate can be used directly to address the GLCD, the `x` coordinate must be converted into a row and bit number. This is done by dividing the `x` coordinate by 8 (i.e. `rrcf` right rotate 3 times) and using the division remainder `n` to set the n^{th} bit. Finally, the routine reads the current value of the GLCD at the corresponding byte coordinate, and performs a bitwise `OR` operation with the remainder-derived byte to turn on the pixel in the case of the head. For the tail, a bitwise `subfwb` subtraction is carried out to turn off the pixel corresponding to the tail.

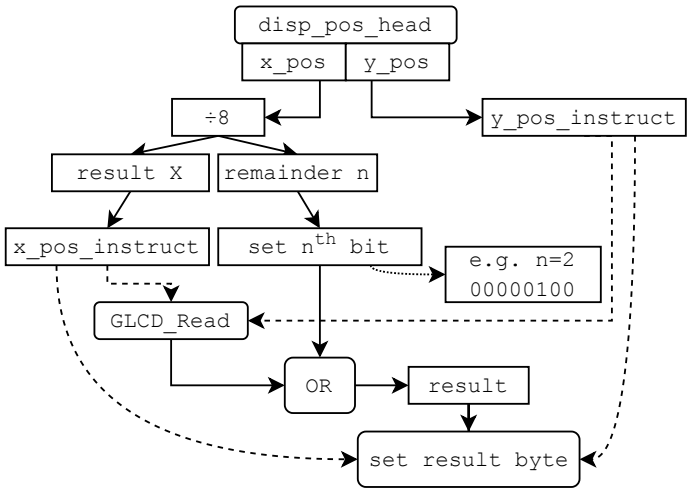

Fig. 4. Flowchart illustrating the display position subroutine for the case of pixel turn-on.

E. Spawn Food

This routine, located in `randgen.s`, is responsible for spawning the food in a random location. Fig. 5 graphically outlines the algorithmic structure of this routine.

The routine starts by reading timers 0 and 1, which are initialised in the initialisation routine, this yields two pseudo-random numbers between 0 and 255. Since a position between 0 and 63 is wanted, both random numbers undergo a bitwise `AND` operation with `00111111` which finally yields two random numbers between 0 and 63 for `x` and `y`. The routine checks that this position does not lie inside the snake (`check_if_in_snake_food`), if it does, it tries

again until a position not in the snake is obtained. Once a valid coordinate is generated, this is stored in `food_x` and `food_y` for use in other subroutines and said position calls the `display_position` subroutine to make the food appear on screen.

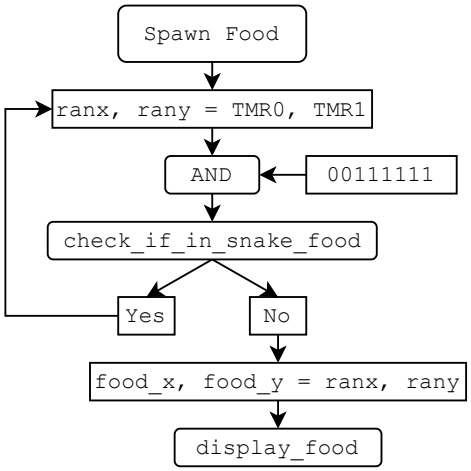

Fig. 5. Flowchart illustrating the spawn food subroutine.

IV. Hardware Design

The alpha prototype built is based on the PIC18F87K22 microcontroller, which is a versatile chip with a number of peripherals and features useful for developing with it—e.g. multiple timers that can run at different speeds for the aforementioned pseudo-random number generation. This PIC18 device is an 8-bit microcontroller with a maximum clock speed of 64 MHz, 128 kB of flash memory and 3.9 kB of RAM. It has 10 timers with varying functionality (of which 2 are used) and a maximum power consumption of 100mW under 64MHz operation in `PRI_RUN` mode [1].

The GLCD used is a 128×64 pixel monochrome display from WINSTAR, which uses a NT7108C controller [2][3]. It communicates with the PIC18 controller in parallel via an 8-bit bus [3]. The pixel response time is of 0.3s and 0.2s for rise and fall times respectively. The maximum power consumption is 380mW [2] (350mW from the backlight and 30mW from the remaining components). This high pixel response time means there are noticeable ghosting artefacts when playing at high speed, which makes the snake appear longer than it is; although an inconvenience, this does not pose a considerable challenge for gameplay or hinder the experience significantly enough as to consider a more performant display. Fig. 7 shows the connections between the GLCD and the PIC18 controller using `PORTB[0:5]` as the instruction bus and `PORTD` for data bus, labelled `RBx` and `RDx`.

The keypad used is a 4×4 matrix keypad with 8 pins: 4 for the rows and 4 for the columns. It is connected to the PIC18 controller via `PORTE` since this is one of the ports that has pull-up resistors, required to operate this particular type of keypad. This is a passive device which has a very low (negligible) power consumption. For gameplay only 4 of the 16 keys are needed; therefore, a 4-key directional pad could be used instead, which could use 4 pins instead of 8. Although a driver rewrite would be required, this would be near-trivial given the complexity of the hardware. Fig. 6 shows the hardware connections between the keypad and the `PORTE` pins.

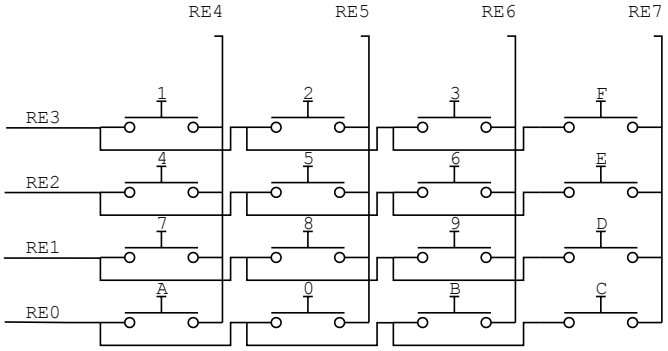

Fig. 6. Hardware schematic of the keypad connection with the `PORTE` pins.

Power Considerations

The power consumption of this prototype is quite low. The GLCD has a maximum power consumption of 380mW, and the PIC18F87K22 has a maximum power consumption of 100mW. The keypad has a negligible power consumption. The total power consumption of the prototype is therefore 480mW, which is well within the power supply capabilities of two AA batteries.

The target battery life is of 10-15h, therefore a single AA battery must be able to deliver 240mW for such an amount of time, considering this is a worst-case scenario. This is consistent with values given by battery manufacturer Duracell, which states a service life of ∼12h for such load [5].

This power source poses a challenge, that is, the voltage range of a single AA battery is 1.5V to 0.8V depending on its charge state. Such voltage is insufficient to power the PIC18F87K22 microcontroller and the GLCD. The solution to this problem is to use a boost converter, which is a DC-DC converter that steps up the voltage from a lower voltage source to a higher voltage one. A proposal boost converter is the TPS613226A from Texas Instruments, which has very suitable characteristics for hand-held consoles like the proposal prototype:

- 0.9 to 5.5V input voltage range [6]
- 2.2 to 5V output voltage range [6]
- 1.8A maximum output current [6]
- About 90% efficiency for relevant operating voltage and current ranges needed [6] (See Fig. 8)
- 6.5µA quiescent current [6]
- A price of $0.13 per unit [6].

Texas Instruments is a well-regarded manufacturer of semiconductor devices and integrated circuits. The TPS613226A is a very suitable product choice due to high efficiency, very competitive price and a number of safety features like overtemperature protection and current limit operation. Such safety features will become essential for obtaining potential regulatory approval for a children-targeted device in some regions.

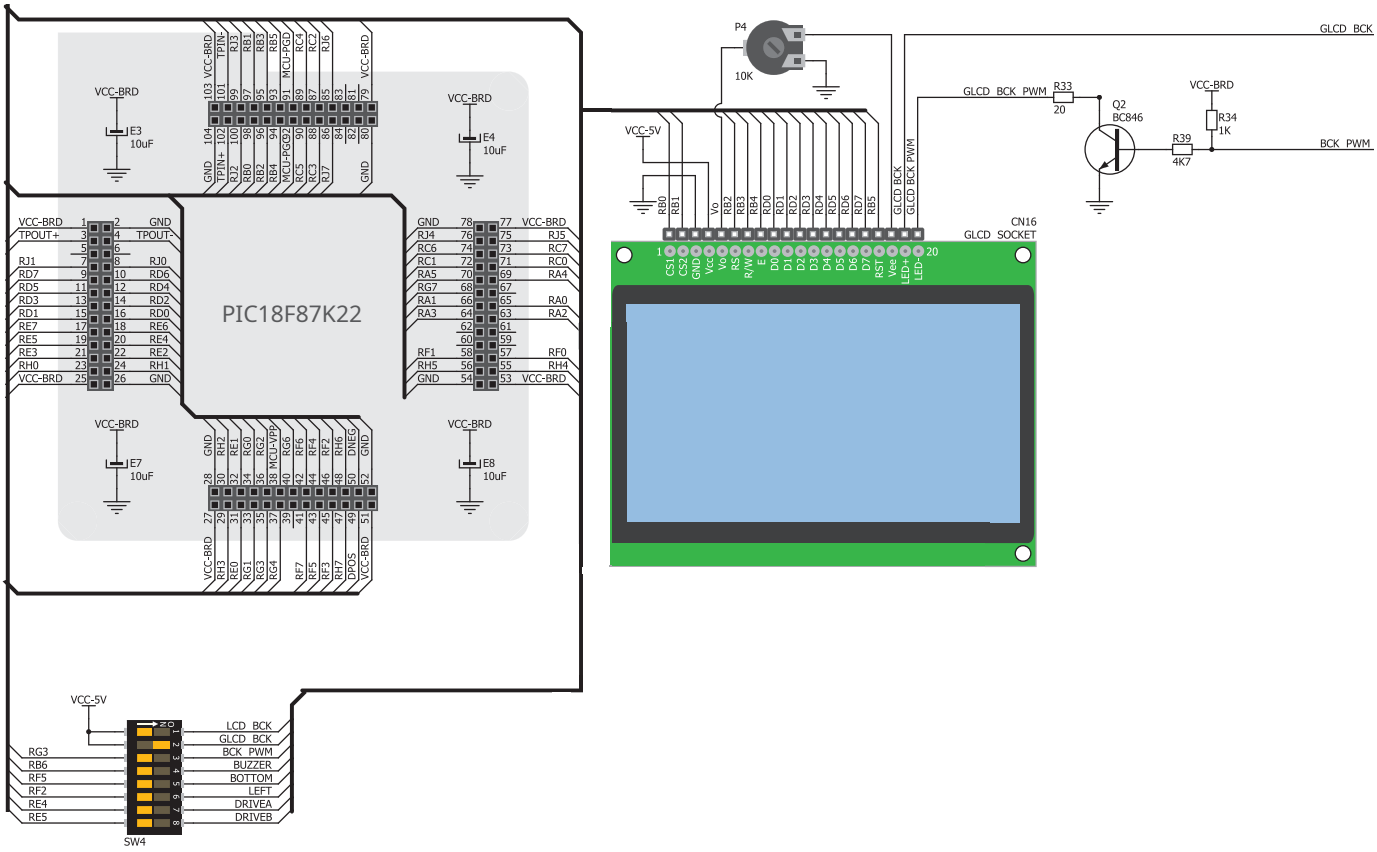

Fig. 7. Hardware schematic of the PIC18 microcontroller with the GLCD connected to it via the PORTD and PORTB ports. Adaptation from [4]

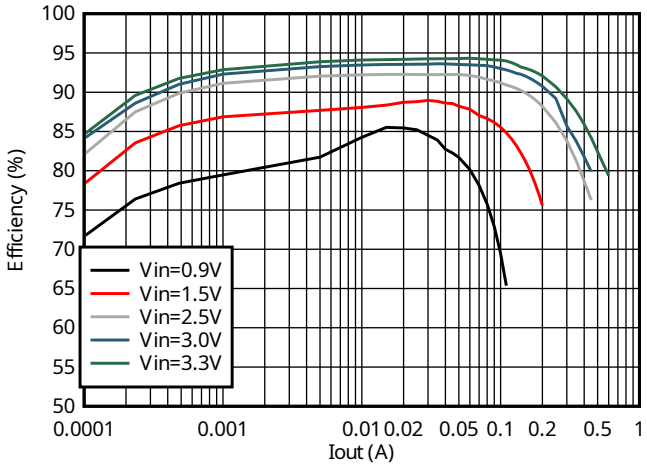

Fig. 8. Efficiency vs. output current graph for the TPS613226A boost converter. Note that the relevant curves are the ones for 1.5V to 3V. Obtained from [6].

After taking an conservative estimate of 90% efficiency for the boost converter, the worse case scenario power draw would be of 530mW, which would still last over 10h [5].

V. Results and Performance

The developed prototype is a fully functional gaming console that meets the initial design goals. It has been thoroughly tested for bugs and performance issues and the following observations are made:

1) **Stability and Bug-Free Operation.** The console operates without any known bugs, ensuring a seamless gaming experience.
2) **Frame Rate and Responsiveness.** Without any delays to purposefully slow the gameplay down, the console is able to achieve a maximum of 13 frames per second, indicative of a very high performance level with no noticeable input lag during gameplay.
3) **Absence of Music.** Due to unavailability of the initially proposed MP3 player module, no music or other sound effects were incorporated into the game in this initial alpha prototype. This will be touched upon in section VI.
4) **Screen Utilisation Limitations.** Due to time constraints in development, the game has only been programmed to run within the left side of the screen, i.e. `CS1` and not `CS2`. A small portion of the game logic must be changed to use both sides of the display: the `y_pos` will accept values from 0 to 127; therefore, values 64 to 127 will have to have a condition that selects `CS2` instead. Alternatively, using a 64×64 display may be considered for future prototypes.

VI. Updates and Improvements

There are several possible improvements to the current implementation of the prototype presented; these range from

addition of new features, to improvements in performance, to code refactoring for a more robust codebase.

A. New Features

The alpha prototype presented showcases the core functionality of the game, but there are a number of user-facing features that could be added to improve overall refinement of the end product. A start or game-over graphic that occupies the entire screen could be added. Perhaps such screen may include the logo or slogan of the customer brand (e.g. McDonald's logo on start-up). The suggested implementation for this would be to write an inelegant solution involving hard-coding display sections instead of looping through a look-up table, i.e. a loop unrolling-like implementation. This utilises the abundance of Flash memory available in the PIC18 microcontroller (currently a little over 1% is used) and would result in an easily implementable solution with maximum runtime performance and minimum development time.

Another feature would be the initially proposed use of music, both in-play music and sounds. This could be easily done using the proposed MP3-decoder with its own music repository inside a microSD card, wherein the microcontroller only sends instructions via UART to select MP3 files on-demand. An extra power consideration regarding music was brought up during development, with the concern that such a device would be a non-negligible power drawing component. However, after investigation, it is seen that this device only consumes about 50mW [7], with a further ∼110mW of power draw using a quality class-D audio amplifier like the Texas Instruments TAS2521 [8][1]. Music implementation is indeed possible with the outlined components, although it will not be a trivial addition to the system. Careful consideration will be required to find an optimal class-D amplifier with the correct specifications and features (e.g. automatic shutdown mode) like the chip provided, and further software-related issues will have to be circumvented as they arise.

B. Performance Improvements

The current implementation of the game is very performant, both in terms of speed (FPS and input lag) and estimated battery life. However, modifications could be made to further increase battery performance. As the main source of power draw originates from the GLCD, more precisely its backlight, an obvious way to increase performance is to be able to regulate the intensity of its backlight. This would require turning on switch 3 from SW4 as seen in Fig. 7 and writing software for the microcontroller to regulate the backlight via pulse-width modulation (PWM). A decrease in display brightness would, theoretically, reduce power draw by up to 350mW [2][2], which would leave a total power draw of 145mW (accounting for DC-DC conversion losses) which would make the console last for over 24h [5].

[1]This assumes a 90% amplifier efficiency, 10% content-dependent factor, and a 1W peak power target

[2]A 350mW reduction would be equivalent to turning off the backlight, hence the theoretical maximum

C. Code Refactoring

It is evident that the current codebase is not production-ready due to a number of factors; nonetheless, these are not significant enough as to impact the functionality of the prototype. There are a number of debugging-related bits of code that should be removed in a refactor. This entails the removal of routines that were initially designated as global solely for debugging purposes that are no longer necessary (as globals).

A number of commented-out lines are present that should be deleted. Lastly, there is some copy-pasted code with slight modifications, who's comments should be changed for consistency with the application they serve. An example of this is `check_if_in_snake_food` from `randgen.s` which is the same routine as `check_if_in_snake` from `array.s` except for one checks whether the food or the new head position lies within the snake's positions to generate a new random position to spawn the food in and the other terminates the game.

VII. Conclusions

Snack&Snake alpha represents a significant stride in interactive entertainment within the fast-food industry. Its robust, efficient design coupled with a potential focus on sustainability aligns well with the evolving corporate responsibility goals of major fast-food chains. Furthermore, it provides an innovative product with potential market value. While the prototype showcases core functionalities and promising performance, future iterations could explore enhanced features and further optimisations. Overall, Snack&Snake emerges as a viable, environmentally-friendly option for on-site entertainment, reflecting a thoughtful blend of technology and commercial viability.

VIII. Product Specifications

- **Microcontroller:** PIC18F87K22
- **Flash Memory (*used*, available):** *1.5kB*, 128kB (1.2% used)
- **RAM (*used*, available):** *548B*, 3.9 kB (14.2% used)
- **Display:** 128×64 monochrome LCD (currently available 64×64)
- **Input:** 4×4 keypad
- **Power Source:** 2× AA batteries
- **Game Speed:** 13 FPS (can be decreased for slower gameplay)
- **Maximum Snake Length:** 256
- **Estimated Battery Life:** +10 hours (with backlight on), +24 hours (with backlight off)
- **Open Source Code:** Yes [9]

Acknowledgment

In the preparation of this report, several diagrams have been utilized which were collaboratively developed with my lab partner, Mihnea Bucur.

REFERENCES

[1] Microchip. (2011) Pic18f87k22 family data sheet. Accessed: January 15, 2024. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf

[2] L. Winstar Display Co. (2009) WDG0151-TMI-V#N00 specification. Accessed: January 15, 2024. [Online]. Available: https://www.farnell.com/datasheets/1878006.pdf

[3] N. S. Ltd. (2014) Nt7108cds datasheet. Online. Accessed: January 15, 2024. [Online]. Available: http://www.neotec.com.tw/upload/products/76/NT7108CDS_Rev1_0_20150817_1.pdf

[4] MikroElektronika. (2013) Easypic pro v7 schematic. Online. Accessed: January 15, 2024. [Online]. Available: https://download.mikroe.com/documents/full-featured-boards/easy/easypic-pro-v7/easypic-pro-v7-schematic-v101.pdf

[5] Duracell. (2022) Duracell Optimum OP1500 AA Battery Technical Data Sheet. Accessed: January 15, 2024. [Online]. Available: https://www.duracell.com/wp-content/uploads/2016/03/OP15US0122-A.pdf

[6] T. Instruments. (2023) Tps61322 product page. Online. Accessed: January 15, 2024. [Online]. Available: https://www.ti.com/product/TPS61322

[7] MikroElektronika. Accessed: January 15, 2024. [Online]. Available: https://download.mikroe.com/documents/datasheets/KT403A_datasheet.pdf

[8] Texas Instruments, "Tas2521 audio codec data sheet," https://www.ti.com/lit/ds/symlink/tas2521.pdf, 2023, accessed: January 15, 2024.

[9] D. W. Falcó. (2023) Snack snake github repository. Accessed: January 15, 2024. [Online]. Available: https://github.com/danifalco/Snack-Snake