



PROJECT - SNAKE

Embedded Systems Group 3

Abstract

A report on an implementation of the popular Snake game on the LM3S1968 Microcontroller kit. The game was successfully implemented well ahead of the provided time frame. Additional advanced features (Different Mazes, Sound, Speeds, Score etc.) were also added for further learning and improving the game.

Zubair Lutfullah KAKAKHEL, Student ID: 200720710

MSc. Embedded Systems | University of Leeds

Table of Contents

Introduction	3
Platform	3
Game Description	4
Design Methodology	5
Top-level Design & Motivation	5
Modular Structure.....	5
Coding Structure	6
Project Structure (Implementation)	7
Initializations	7
SysTick Timer	7
Global variables.....	7
GPIO Input approach.....	8
Direction Update and Pause game	8
Display Init and Usage.....	9
Snake Structure	9
Menu Mode	10
Game Mode	11
Collision	12
Food Ingestion.....	13
Snake Update (Movement of Snake)	15
Update Display	17
Integration	17
Notes on Additional Features	17
Score	17
Background Sound	17
Maze.....	18
Variable Speed	18
Screenshots of Gameplay	18
Concluding remarks	19
Appendix	19
Main.c	20
My_init.c	20
Snake.c	23
My_init.h	30
Snake.h	30
References	31

Table of Figures

Figure 1 Diagram showing features of the LM3S1968 Kit	3
Figure 2 Snake game intro	4
Figure 3 Snake gameplay	4
Figure 4 Snake Game Top-level flowchart	4
Figure 5 Code flow Top level Flowchart.....	6
Figure 6 Flowchart showing what runs when an interrupt is received from the GPIO port	8
Figure 7 Overview of Code flow inside Game Mode	11
Figure 8 Flowchart showing algorithms which check if Snake collided	12
Figure 9 Flowchart showing algorithms which check if Snake ate food	13
Figure 10 Flowchart showing code that updates the snake elements array	15
Figure 11 Flowchart showing the update display routine	17
Figure 12 Screenshot of game without a maze. Borders only	18
Figure 13 Screenshot of game with a maze	18
Figure 14 Screenshot of game with another maze	18
Figure 15 Screenshot of Menu	18
Figure 16 Checking the random fruit generation.....	18

Introduction

In this report, we discuss an implementation of the popular Snake game on the LM3S1968 Microcontroller Kit. The LM3S1968 Microcontroller kit is highly suitable for this task as it has an interfaced OLED Display and on-board buttons positioned in the Up/Down/Left/Right configuration. The processing capabilities of the board and the ARM Cortex M3 core are more than sufficient for the task at hand. The OLED Display is 128*96 which gives a good resolution and the size is larger than most old Nokia handsets that would play the game. To give a comparison, the popular Nokia 3310 termed as an indestructible phone by some, has a screen resolution of 84 x 48 pixelsⁱ. It included version 2 of the Snake game.

We were allowed to freely choose the level and complexity of the implementation of the game. A modular approach was used to create a good structure of the code. This allowed additional features developed later on to be integrated. It was intentionally decided to not take any sort of help regarding implementation of the snake game from the internet. This enabled an enhanced learning process throughout the project.

The following two sections give an introduction to the platform and the game itself. After that, the design methodology for the actual implementation is discussed. The “Design Methodology” section gives the level of the game intended to be implemented. It was decided to implement the core functionality (movement, fruit, collision, growth) first. Afterwards, additional features (menu, sound, maze, score) would be implemented. The next section gives is the “Project Structure”. As the title indicates, it gives a little more detailed overview of the project and its structure before going into detail. The sections after that provide an in-depth analysis of the code along with flowcharts explaining the flow of the game. The flow is separated into a “Menu Mode” and a “Game Mode”. A few screenshots of gameplay and testing are given after that. Finally, the conclusion summarizes the goals achieved during the project implementation. At the end of the report, the appendix gives the entire code along with a short explanation of the files. References are listed on the last page after that.

Platform

The LM3S1968 Microcontroller Kit has an ARM M3 core based microcontroller. The ARM-M line of cores are specifically intended for low power microcontroller implementations. The figure belowⁱⁱ shows some of the on-board features available on the kit.

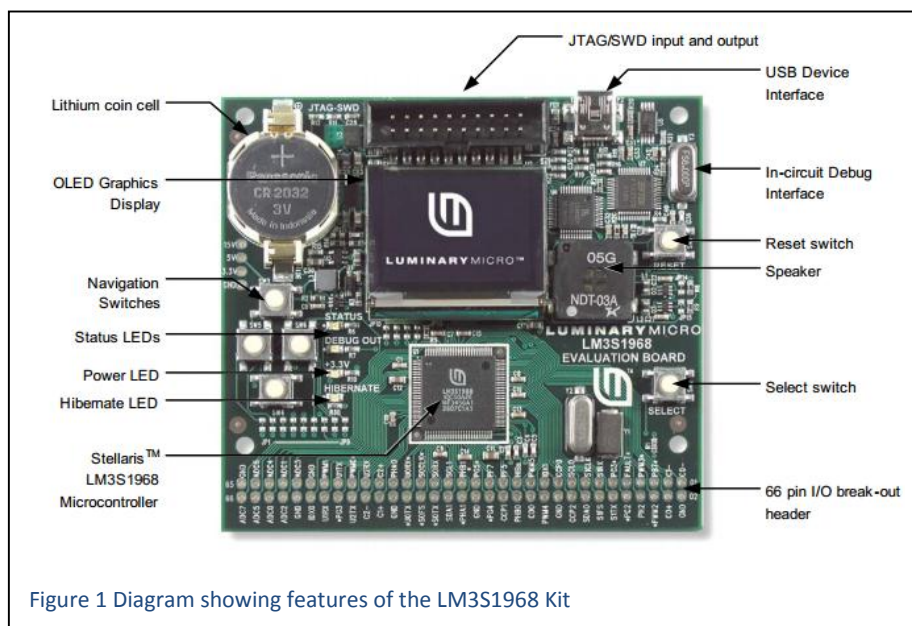


Figure 1 Diagram showing features of the LM3S1968 Kit

The on-board peripherals that were used were the OLED Display to display the game. The navigation switches were used in a standard Up/Down/Left/Right configuration. The Select switch was also used to pause the game or to start the game. The Status Led was also blinked showing the game is live. The on board In-circuit Debug interface via USB was used to debug the code at various stages. The speaker was used for in-game audio.

Apart from the hardware interfaces, the M3 core provides the Systick timer and external hardware interrupts. Both were used extensively in implementation.

Game Description

The Snake game was first seen in the 1970sⁱⁱⁱ. However the game gained significant popularity once it was included in Nokia Handsets. The figures below show the second generation Snake game running on the Nokia 3310.

In this game, the player is in control of a thin snake. The main objective of the game is to navigate the snake to eat food that appears randomly on the map. As the snake eats the food, the snake grows longer. Points are awarded when it eats the food. The game ends if the snake dies. The snake dies if it eats itself (i.e. collides with its own body which can grow pretty lengthy). Or if the snake collides with the boundary wall. The speed of the snake can be changed. A faster speed results in a higher number of points awarded to the player on every food item eaten.

Variations of the basic game (seen in Snake II) include introducing a maze in the map. Any collision with the maze also causes the snake to die. Another popular variation in the random appearance of a bigger food item for a short duration. This bonus item gives a significant amount of points to the player.

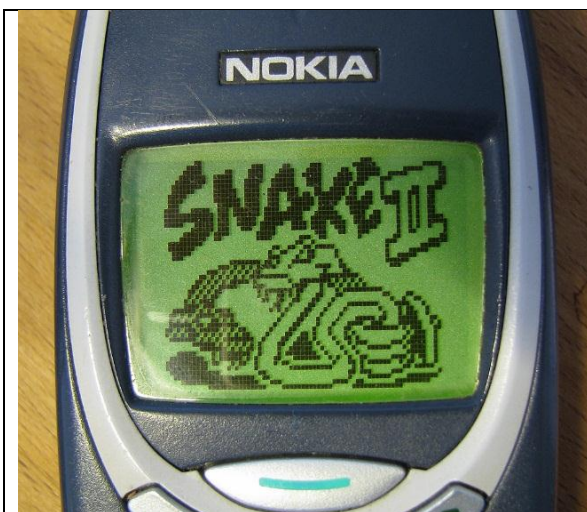


Figure 2 Snake game intro^{iv}

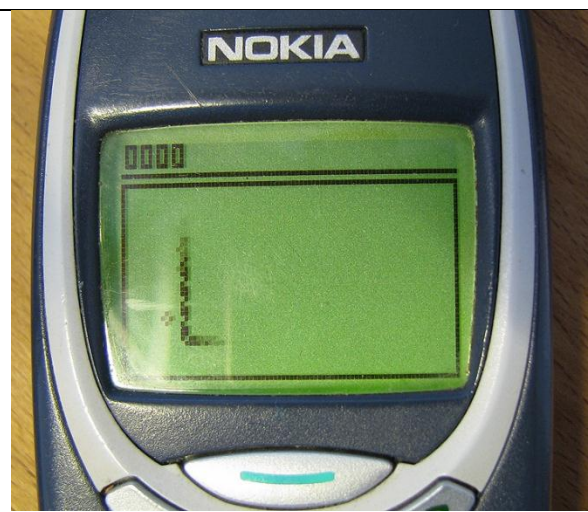


Figure 3 Snake gameplay^v

The basic flow of the game can be modelled in a simple looping flowchart.

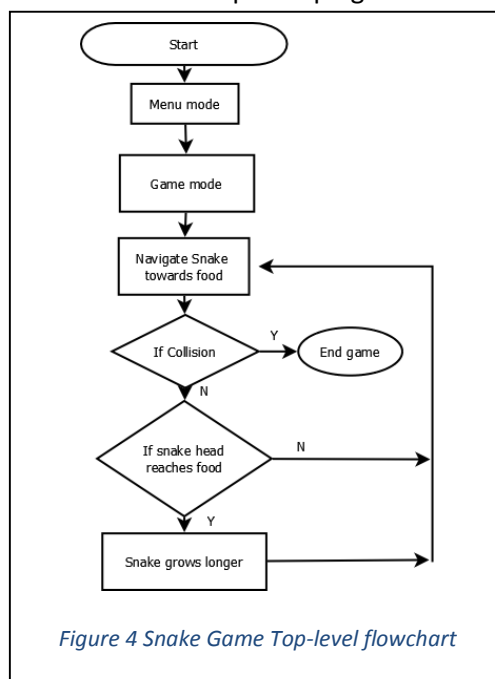


Figure 4 Snake Game Top-level flowchart

Design Methodology

Snake has simplistic gameplay with a primitive looking display component. However, a little thought leads to the conclusion that the variables of the game are not as simple as they initially seem. Also, the requirements of the project were not specific regarding the level and complexity of the games implementation. Instead of coding aimlessly, a significant amount of time was invested in chalking out specific requirements of the game and then designing a stage-wise implementation structure around it. The following section gives more detail regarding the planning that went into this design. After that, the actual breakdown and coding structure is presented.

Top-level Design & Motivation

Given the complexity of the task at hand, the project was broken down into multiple stages and chunks to make it manageable and encapsulate one specific feature of the game. The integral functionality of the game was to be the first stage of implementation. It was separated from additional features such as extra mazes, sound etc. This first stage would allow us to achieve a significant threshold of completion in the project. The second stage would be the additional components which would be layered on top afterwards. The framework and the code structure would allow such additional components to be easily added to the project.

The following sections give a breakdown of various modules within these stages and insight into the coding structure that allowed the implementation in such a manner.

Modular Structure

The core functionality of the game can be further broken down into separate modules as seen below:

1) Movement of the snake.

Along with basic movement. The entire snake has to move in a manner as the player has commanded previously. If the player commanded the snake to move in a zigzag manner, the rest of the snake body has to follow the sequence of the inputs.

2) Navigation of the snake.

On the press of the key, the snake should change direction. The response is the key here to ensure effective gameplay. Another aspect to note here is that the snake cannot change into the opposite direction immediately. If the snake is facing downwards, the down and up keys are useless. The snake can only change direction at right-angles to its heads' current bearing.

3) Food

It has to be checked whether the snake has eaten the food. If eaten, another food item has to be generated at a random location. A few key points learned later on during development were that the generation of the food location has to be such that the snake is not currently occupying that position. Another factor is that in case there is a maze in the map, the food generated does not have to overlap the maze. These key features result in a slightly complex algorithm that generates the food.

4) Collision detection

At every step of the snakes' movement, it has to be checked whether the snake has collided with either the wall or itself. Both these situations end the game. The collision gets especially tricky when mazes are involved.

Apart from the core components that make up the game, some additional components were implemented to enhance the level of the implementation:

1) Menu system

The menu is the basic settings where the player can choose the speed of gameplay, volume of the audio and the maze.

2) Sound

Audible responses to the snake eating food and some background music in the intro section enhance the experience of the game itself. One major problem noted in the generation of the sound is its relationship to the SysTick timer. The SysTick timer is used to change the speed of the snake. But the timer is also linked with that of the sound generation. Sound plays faster/slower corresponding to the SysTick timer period.

3) Maze

Different mazes add an extra fun factor to gameplay. Their implementation increases the complexity of several aspects of the code.

4) Varying speeds

Various speeds allow players of every level and age to play the game.

5) Score

No game is complete without the competitive psychological factor induced by the scoring system.

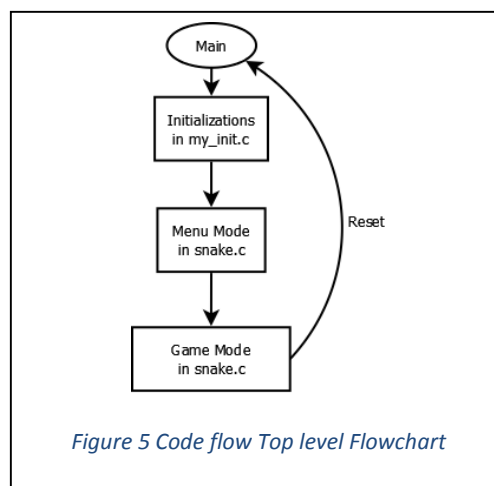
Coding Structure

Before starting with any sort of implementation, a lot of thought was given to the base structure of the entire code which is the most vital part of any program. All these components require that the base structure be exceptionally sound as the final code has to involve all these parts. In higher level languages, there is great flexibility with object oriented programming practises. At this level, we only have C at our command and the modularity of the code is in our hands. Any coding in any component has to be written with an understanding of the rest of the modules in the back of the mind as well.

Initializations are lumped together in one file (my_init.c). These include clock, port, interrupt, variables etc. Both the menu and gameplay are periodic and handled in the SysTick Interrupt. A global variable differentiates between both modes.

As part of the lab exercises, it was learned that the SysTick timer is essential to periodic implementations. The movement of the snake is by nature periodic. The speed of update depends on the period of the timer. The direction of the next update depends on the last button pressed. This led to the conclusion that the gpio interrupts should update a current direction variable. While the SysTick timer would handle the rest of the values to update. This approach led to an elegant solution which allowed significant modularity. Hence, the SysTick interrupt routine is used to call other functions. Both the functions and the interrupt routine are handled inside 'snake.c'.

A top level flow chart conforming to this design is given below. It shows the file structure with reference to the two modes of the game.



The next section gives further detail on the implementation structure.

Project Structure (Implementation)

Before diving into the heart of the game itself, some background on the project structure is necessary. This sections give a little detail regarding the initializations, SysTick Timer configuration, Global Variables, GPIO interrupt, Display and the Snake Structure used to represent the snake in the program memory.

Initializations

The self-explanatory code is given below. Details of Init functions are in the appendix. Most were initialized to default values used in previous lab tasks.

```
Clock_Init();
PortG_Init();
Sound_Init();
OLED_Init();
SysTickInt_Init();
IntMasterEnable();
```

Systick Timer

The Systick timer configuration requires more explanation. In any game, there is a periodic update of all aspects of the game and the display. For this purpose the Systick Timer is used as a periodic timer. The Systick interrupt routine is the main periodic update of the elements of the game. It is initialized using the clock of the oscillator with dividers. This ensures that the timer remains consistent even if the clocks are changed. Thus the code portability is increased.

An interesting application of the Systick Timer was found when the speed of the snake is changed in stage two of the development. Changing the speed of the snake means increasing the frequency of the periodic updates. Hence, to do that, we simply reconfigure the Systick using the clock of the oscillator but with a different divider value corresponding to the one taken as an input by the player using the menu.

Global variables

Global variables are not recommended in modern structured programming. In some companies, adding a global variable to the code requires a written request to the supervisor providing explanation and reasoning behind it. However, that is usually in the case of extensive higher level languages and large scale implementations. Also, in higher level languages where the code is running on top of an operating system, there are several considerations regarding the usage of memory. In embedded systems, specifically in micro-controllers, the practice is not as stringently held as the programmer is usually partitioning the memory himself.

Therefore, in our implementation, global variables are freely used as needed. The following variables were used. The comments explain the purpose of each variable.

```
// Global Variables
unsigned char Map_Display[Yres][Xres]; //2D map of the display
unsigned char Frame_Display[Xres*Yres/2]; //Linear array input required by Display Lib
struct Snake My_Snake[3072]; //Upper limit of snake length for a very pro player
int Current_Snake_Direction; //Current direction of the head of snake
int Snake_Tail_Index; //Tail index for location of tail in My_Snake
int FruitX; //X coordinate of Current fruit location
int FruitY; //Y coordinate of Current fruit location
int Pause; //Game in Pause/Unpause mode.
int Score = 0; //Score counter
int Speed = 1; //Speed of snake. Related to divider of Systick
int Volume = 5; //Volume of the sound
int Maze = 1; //Maze selection. 1,2,3 possible.
int Game_Mode = 0; //Game/Menu mode selector.
```

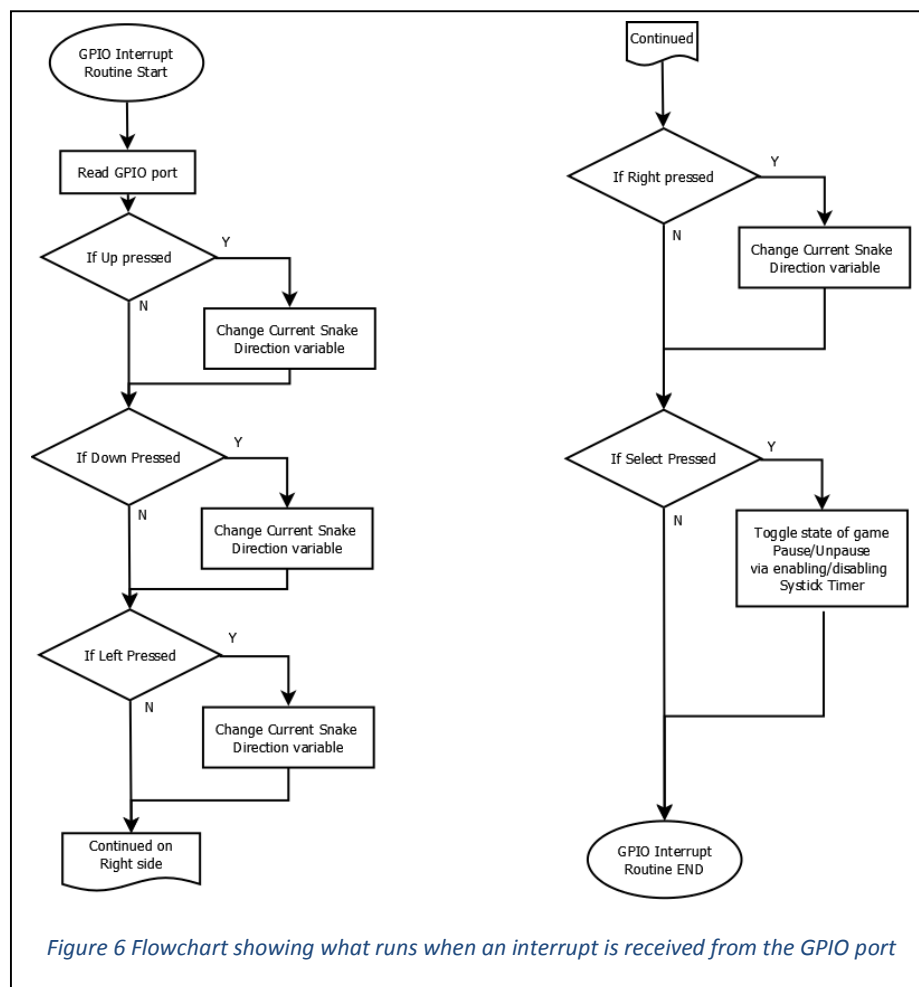

GPIO Input approach

The navigation of the snake requires the usage of input buttons to change the direction of the snake. Apart from that, the menu configuration also requires input. A hybrid of Gpio interrupts and the SysTick Timer is used to update the direction of the snake. The GPIO interrupt is used to change a global variable indicating the current direction of the head of the snake. The SysTick Timer interrupt routine is used to periodically check the state of the global variable. In this case, bouncing is not an issue either as the actual variable changed is polled in the SysTick timer. Also, switch bounce effects are visible when a consecutive sequence of high/low of the same input (i.e. switch bounce) is relevant. In this case, if consecutive inputs of the UP direction are detection. The global variable is simply updated with the same value. Therefore, switch bounce does no harm to the system.

There are alternative techniques possible to acquire input from the player. Bitbanding could have been used to check the state of the input in one cycle using the memory mapped region. That would have been a more optimized solution. However, the current approach leads to a more portable code. The snake game does not tax the resources of the microcontroller to validate the effort of optimization. The next sections gives slightly more detail regarding the input related to navigating the snake. The Menu mode input is a simple polling routine inside the SysTick Timer interrupt.

Direction Update and Pause game

As discussed previously, the direction of the snake is updated in the GPIO Interrupt routine. This routine also allows for pausing the game in the middle. The following flowchart gives a detailed overview of what happens inside the routine. The basic crux is that there is a global variable that gives the current direction of the head of the snake. The GPIO routine updates this variable. The actual check on the global variable happens in the periodic SysTick interrupt routine.



The code a simple sequence of if/else checks. It can be seen in the Appendix and is not reproduced here.

Display Init and Usage

The OLED display is divided into two sections. The bottom section displays the score as the game progresses. The top section is where the game is running. Apart from that, for the purpose of better gameplay and easier visibility the snake sections and the fruit are 2x2 pixels instead of 1 pixel. This creates a measurable size and allows the player to easily navigate the snake to eat the fruit. A 1 pixel wide snake with a 1 pixel wide fruit did not allow the player to effectively judge if the snake was moving in the line of the fruit. In that case, the fruit would usually be missed by one pixel.

Snake Structure

In the game, the snake grows longer. The snake consists of sections or elements. At an update, the head of the snake proceeds in the current direction. All the rest of the snake elements move in their respective directions. This requires that each element has to remember the next direction that it has to take. And the direction of the elements will propagate down the elements till the tail of the snake. This is coded using an array of the following structure.

```
struct Snake
{
    int Direction;
    int LocationX;
    int LocationY;
};
```

Each element of the array is a section of the snake. It knows the direction it has to move. And each section has an (X,Y) coordinate pair giving its location. Ideally the snake can be constructed using only a set of directions and backtracking the pixels. However, that would uselessly increase complexity of the entire code structure. Limited Memory is not that critical of a design factor at this level of development.

The array size is fixed to the maximum limit any player can reach if the snake is covering the entire screen. In this case, the tail of the actual snake would simply be an index of the array indicating the last section is stored in that structure. The benefit of using an array of these structures is that the growth of the snake becomes a simple matter of adding an actual element at the tail of the array.

The next section starts diving into the actual game itself. The Menu Mode is presented first. After that, the Game Mode is explained in detail.

Menu Mode

The menu is incorporated inside the SysTick interrupt routine. It is a simple single interface menu. The Up/Down keys change the volume. The Left key cycles through various speeds. The right key cycles through various mazes. Pressing the Select button starts the game.

In this simple instance, a flowchart was not required to show the concept. The code is reproduced below which shows the menu system.

```
void Menu(){
    char buffer[20];
    RIT128x96x4StringDraw("Press Select to Play",5,Yres-18,15);

    RIT128x96x4StringDraw("Up/Down for Vol U/D:",5,Yres-38,15);
    sprintf(buffer, "%d",Volume);
    RIT128x96x4StringDraw(buffer,100,Yres-28,15);

    RIT128x96x4StringDraw("Left to cycle speeds",5,Yres-58,15);
    sprintf(buffer, "%d",Speed);
    RIT128x96x4StringDraw(buffer,100,Yres-48,15);

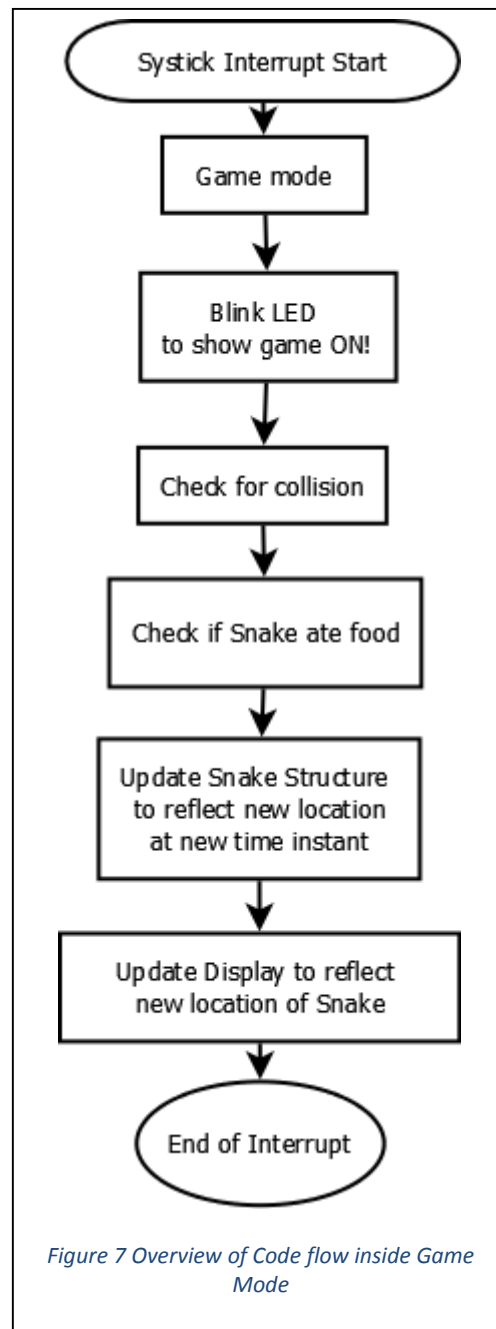
    RIT128x96x4StringDraw("Right to cycle mazes",5,Yres-78,15);
    sprintf(buffer, "%d",Maze);
    RIT128x96x4StringDraw(buffer,100,Yres-68,15);

    if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_UP)) == 0) {
        Volume = Volume + 1;
        if (Volume > 10)
            Volume = 10;
    }
    if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_DOWN)) == 0) {
        Volume = Volume - 1;
        if(Volume < 0)
            Volume = 0;
    }
    if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_LEFT)) == 0) {
        Speed = Speed + 1;
        if (Speed > 5)
            Speed = 1;
    }
    if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_RIGHT)) == 0) {
        Maze = Maze + 1;
        if(Maze > 3)
            Maze = 1;
    }
    if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_SELECT)) == 0) {
        SysTickIntDisable();
        SysTickDisable();
        Game_Mode = 1;
    }
    if(Game_Mode == 1)
    {
        Diplay_Maze_Init();
        Snake_Init();
        GPIOInt_Init();
        SysTickPeriodSet(SysCtlClockGet()/(Speed*20));
        SysTickEnable();
        SysTickIntEnable();
    }
}
```

The code above is simple to follow. It runs at every SysTick Interrupt when the code is in Menu Mode (Game_Mode Variable=0). When the player presses SELECT, the Game Mode is initialized. The next section explains Game mode.

Game Mode

The Game mode is implemented as part of the SysTick Interrupt routine. Every interrupt updates the game. A top-level flow chart for the game mode is given below. Each block in this flowchart is composed of subsections. The subsections are individually explained later in the report.

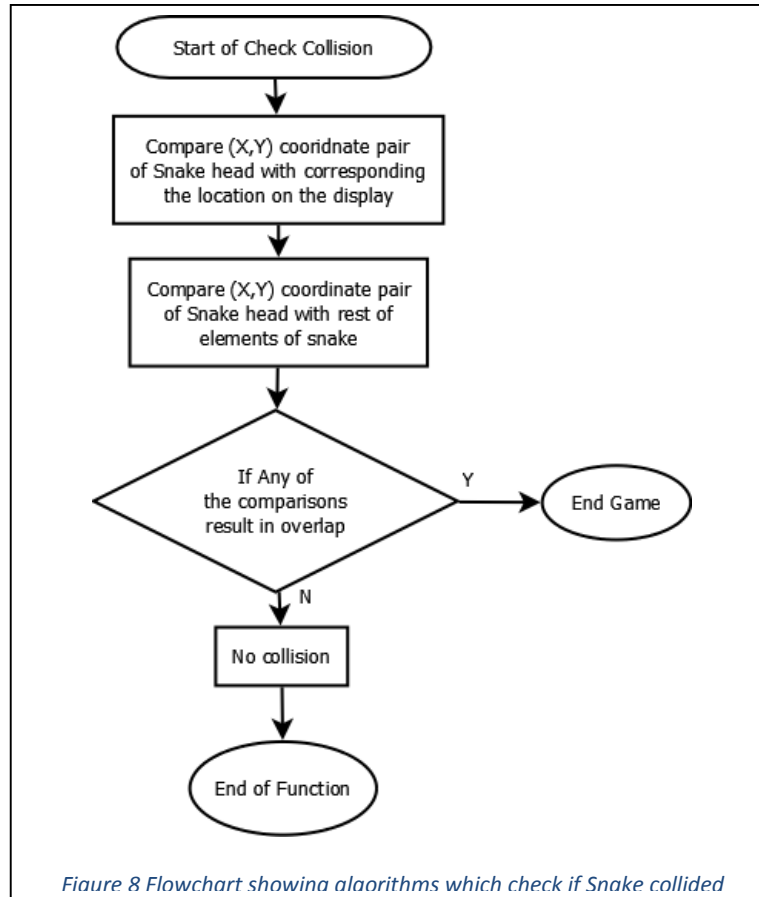


The snake starts off in the middle with a piece of fruit somewhere on the map. After that, each game time-period is one SysTick interrupt routine. Hence, all the above operations are carried out every time instant. The LED is blinked signifying the game is actively running. The snake head is checked if it collided with itself or a wall. The snake heads' location is checked if it overlaps the fruit. If it does, another fruit is generated and the snake grows longer. The snakes' locations and directions are updated in the array of snake element structures. The display is updated to reflect the new location. This process continues until the snake collides with a wall or itself and the game ends.

The following sections explore the top-level blocks in the flowchart in detail.

Collision

In the game, the snake dies if it hits the wall of a maze or any part of itself. The collision is checked using a simple comparison with the location variables of the snakes head with the wall and other snake elements locations. If the comparison is true, the Systick timer ends signalling the end of the game. The game can then be restarted using the reset button. A continuous mode of gameplay would simply require re-initializing the menu mode of the game. However, it has been left in favour of the reset button. The following flowchart gives a top level overview of the steps required in checking for the collision of the snake.



The following code gives an efficient implementation of the flowchart.

```
void Check_Collision(void){
int Map_Collision_Check , Snake_Collision_Check , n;
Snake_Collision_Check = 0;

Map_Collision_Check = Frame_Display[My_Snake[0].LocationY*64 +My_Snake[0].LocationX/2]
+ Frame_Display[(My_Snake[0].LocationY+1)*64 + My_Snake[0].LocationX/2]
+ Frame_Display[My_Snake[0].LocationY*64 + (My_Snake[0].LocationX+1)/2]
+ Frame_Display[(My_Snake[0].LocationY-1)*64 +My_Snake[0].LocationX/2]
+ Frame_Display[My_Snake[0].LocationY*64 + (My_Snake[0].LocationX-1)/2]
+ Frame_Display[(My_Snake[0].LocationY+1)*64 + (My_Snake[0].LocationX+1)/2]
+ Frame_Display[(My_Snake[0].LocationY-1)*64 + (My_Snake[0].LocationX-1)/2];

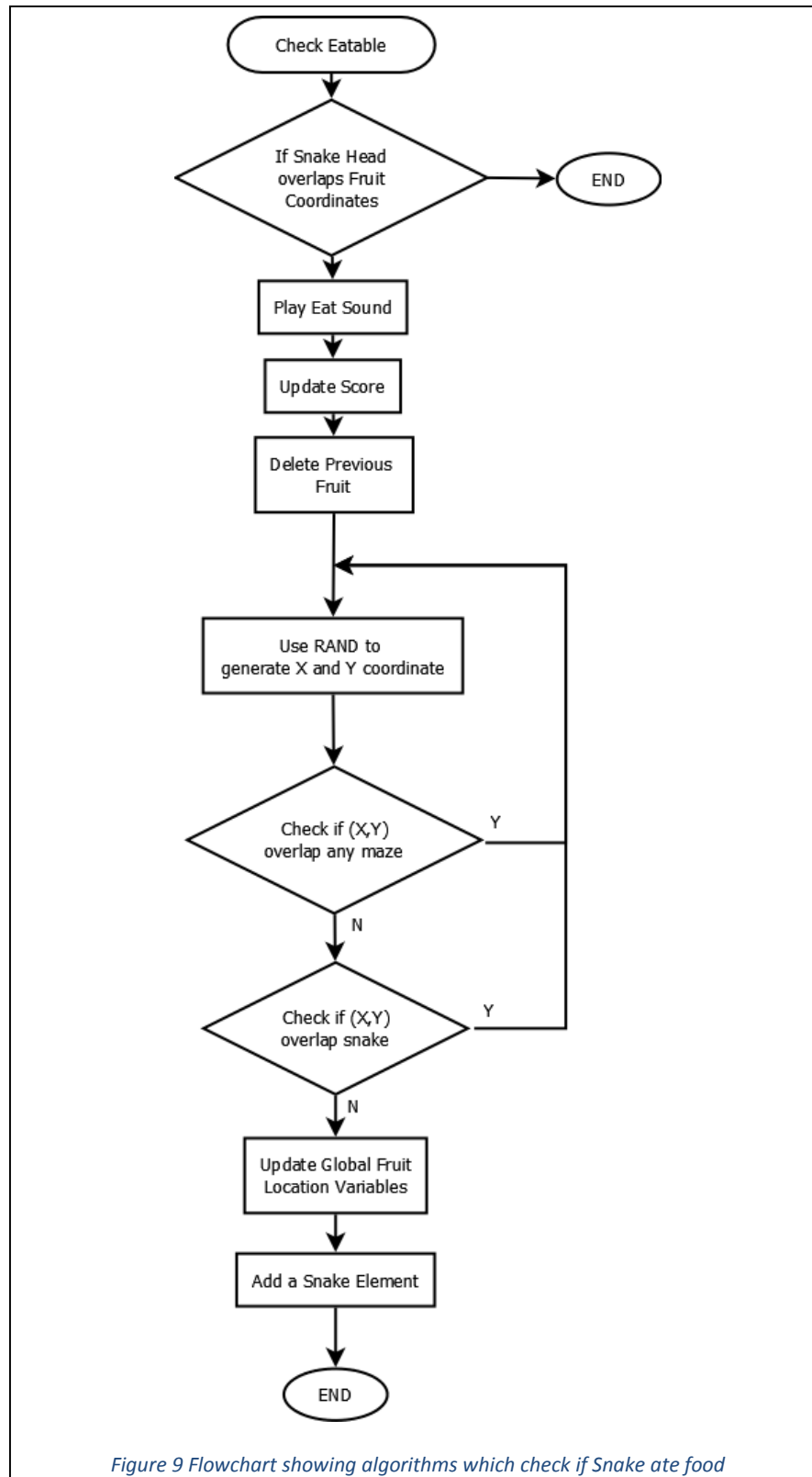
for (n = 1 ; n < Snake_Tail_Index ; n++){
    if( (My_Snake[0].LocationY == My_Snake[n].LocationY) && (My_Snake[0].LocationX ==
        My_Snake[n].LocationX))
        Snake_Collision_Check = 1;
}
if(Map_Collision_Check || Snake_Collision_Check )
    SysTickDisable();
}
```

Food Ingestion

When the player successfully navigates the snake towards a food item, two things happen.

- 1) Another food item appears
- 2) Snake grows longer

These two processes are handled in the Check_Eatable function. The in-built C library for the RAND function is used to generate the random location of the fruit. The flowchart on the next page gives an overview of the entire process.



Adding a snake element can be tricky. But it is easy with the help of the global variable giving the location of the tail of the snake in the array. A Snake section is added at the tail of the snake. A section of the code is given below:

```
void Check_Eatable(void){
    //Check if snake head ate the fruit
    if(
        ((My_Snake[0].LocationY == FruitY) || (My_Snake[0].LocationY + 1 == FruitY) ||
        (My_Snake[0].LocationY - 1 == FruitY)) &&
        ((My_Snake[0].LocationX == FruitX) || (My_Snake[0].LocationX + 1 == FruitX) ||
        (My_Snake[0].LocationX - 1 == FruitX))
    )
    {
        Score = Score + Speed;
        Update_Score();

        Generate_Fruit();

        My_Snake[Snake_Tail_Index].Direction = My_Snake[Snake_Tail_Index-1].Direction;
        My_Snake[Snake_Tail_Index].LocationY = My_Snake[Snake_Tail_Index-1].LocationY;
        My_Snake[Snake_Tail_Index].LocationX = My_Snake[Snake_Tail_Index-1].LocationX;

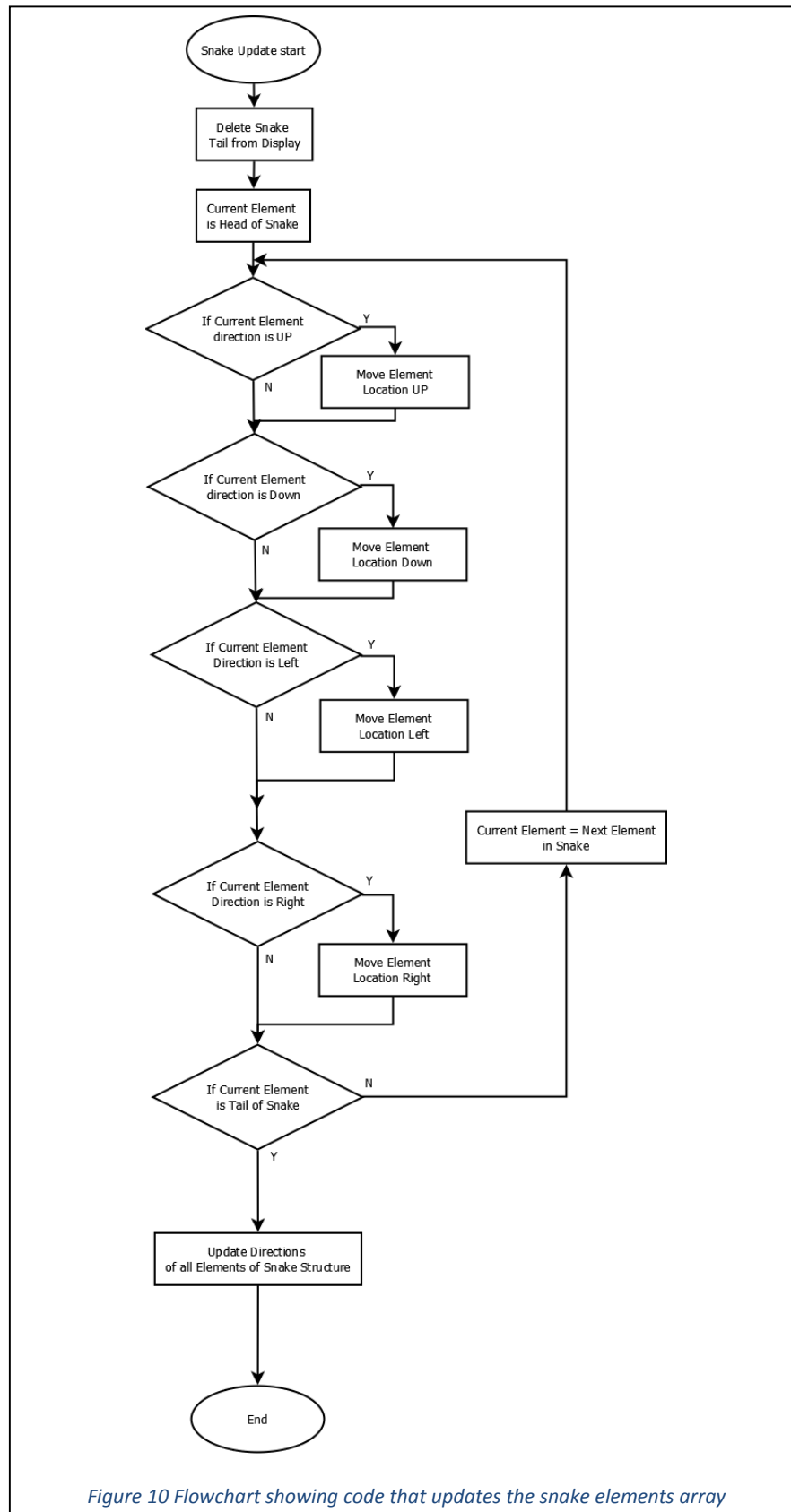
        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_UP)
            My_Snake[Snake_Tail_Index].LocationY = My_Snake[Snake_Tail_Index-1].LocationY + 1;
        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_DOWN)
            My_Snake[Snake_Tail_Index].LocationY = My_Snake[Snake_Tail_Index-1].LocationY - 1;
        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_LEFT)
            My_Snake[Snake_Tail_Index].LocationX = My_Snake[Snake_Tail_Index-1].LocationX + 1;
```

Generating the food or fruit is separated and performed in another function. A tricky part of the food generation is that the RAND function can be given limits. But not specific locations to avoid. Hence, the fruit location has to be checked with that of the maze and the snake as well. If the location overlaps with any wall or the snake, the food is regenerated. The process continues until the randomly generated location is clear of any obstruction.

The function called to generate the fruit is given in the appendix. The checks for ensuring the generated food is not overlapping the maze or the snake result in a lengthy code. Hence, it is not reproduced here.

Snake Update (Movement of Snake)

The update routine of the snake is responsible for every periodic update of the location of the snake i.e. the movement of the snake is handled by this block. Once the entire array of elements of the snake are updated, the new location of the snake is updated on the display while leaving the rest of the display intact. Refreshing the entire screen would have uselessly wasted resources. Hence, an efficient approach is used where the head of the snake is updated and the tail of the snake is removed from the display. This means that for each update few display update operations are needed. The following flowchart gives an excellent overview of the process.



The routine starts by updating the head of the snake and proceeding towards the tail of the snake. First, each elements' location is updated corresponding to its direction. After that, the direction of each element of the snake is updated with the previous elements direction. The process continues till the end of the tail.

The code is reproduced below to show the implementation. The code turns out to be quite simple in its implementation.

```
void Update_Snake(void){

    int n;

    //Delete Tail of snake
    const unsigned char AntiBlob[] = {0x00 , 0x00, 0x00, 0x00};
    RIT128x96x4ImageDraw(AntiBlob,My_Snake[Snake_Tail_Index-1].LocationX,
My_Snake[Snake_Tail_Index-1].LocationY,2,2);

    //Loop to update snake structures current position and next directions
    for (n = 0 ; n < Snake_Tail_Index; n++){

        if(My_Snake[n].Direction == DIRECTION_UP)
            My_Snake[n].LocationY = My_Snake[n].LocationY - 1;
        if(My_Snake[n].Direction == DIRECTION_DOWN)
            My_Snake[n].LocationY = My_Snake[n].LocationY + 1;
        if(My_Snake[n].Direction == DIRECTION_LEFT)
            My_Snake[n].LocationX = My_Snake[n].LocationX - 1;
        if(My_Snake[n].Direction == DIRECTION_RIGHT)
            My_Snake[n].LocationX = My_Snake[n].LocationX + 1;

    }

    for (n=Snake_Tail_Index ; n > 0; n--){

        My_Snake[n].Direction = My_Snake[n-1].Direction;

    }

    My_Snake[0].Direction = Current_Snake_Direction;

}
```

Update Display

In every game, the display for the player is refreshed after the game environment (location of snake, points etc.) have updated. In our case, the main task is to update the location of the snake. As indicated previously, the approach used is to refresh only the pixels of the display where the snake lies in. The previous tail location is already deleted in the Snake Update function because the coordinate location of the tail is localized in that function. In this function, only the rest of the snake is updated.

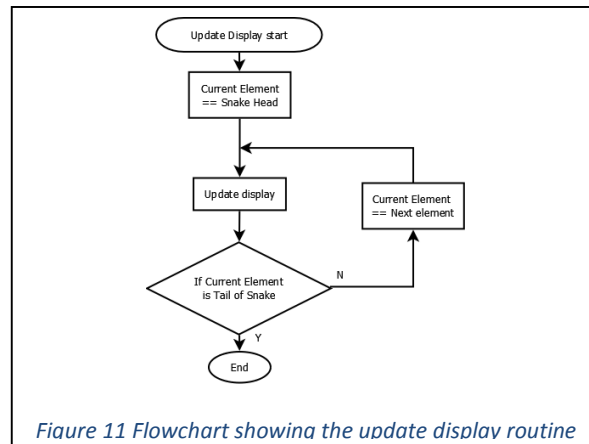


Figure 11 Flowchart showing the update display routine

Integration

All the above components are integrated together in a cohesive manner that result in a functioning game. There are smaller aspects of the code that are not discussed in the report. Documenting the entire code in a stepwise manner is not feasible. However, the code itself is written in an extremely self-explanatory style. All variable and function names make it clear regarding the purpose. Any programmer should be able to follow the code given in the appendix without much trouble.

Notes on Additional Features

The stage two implementations of additional features resulted in several additions to the code at various points. The scoring system was implemented in a simple fashion with a global score counter. The background sound was also added to the project by simply calling a few functions from the Class-d.c library. The maze was the only part that required significant work. The variable speed was achieved by modifying the SysTick Timer period. The following sections give slightly more detail on these additional areas.

Score

To implement the scoring system, a global variable is updated each time the snakes eats a piece of food. The score added is related to the current speed mode of the game.

Background Sound

Background music is common to almost every game since the first few games. A simple repetitive beat running in the background. A significant attempt was made to squeeze background music into the game. However, licencing restrictions prevented linking the sound data to the compiled output. As a result, the sound implementation was restricted to a short sound played every time the player eats the food. And a short intro sound sequence for the menu.

The class-D amplifier library was used in the implementation. Significant help from the example provided with the LM3S1968 kit was taken. The most difficult part was getting the audio library set up. There was no significant documentation found for the library. The basic initializations would not work. After a step-wise debug session, it was found that the library would cause the code to go into the default interrupt handler. That was

undefined and hence the code would appear to hang in an infinite loop. After a while, it was found that the audio library used the PWM interrupt. This was added to startup.s and the audio started working.

Several attempts were undertaken to convert regular music into a compatible format using Audacity^{vi}. However, the audio would be barely distinguishable after conversion. The actual music from the example game with the LM3S1968 kit was taken and cropped to a length that would fit the licensing requirements.

Maze

A maze increases the complexity of the game and thereby enhances gameplay for experienced players. The tricky part was generating the maze itself with respect to the 4-bit display. Instead of a simple 2D matrix linearized, the 4-bit nature of each pixel resulted in a complex linear character array. A software available on the e2e forums^{vii} was used to generate the array for the maze. The software required a bmp image as an input and generated the output array in the correct format. A few simple game displays were created in MS Paint and used in the software. Once the character array was created for the display, the display output was a simple task of calling it from the library.


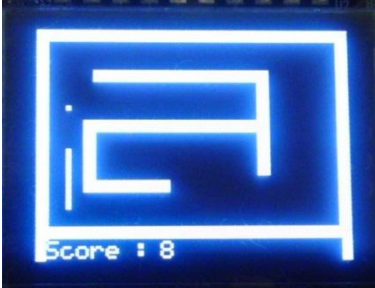

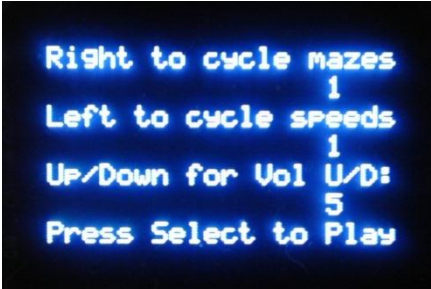
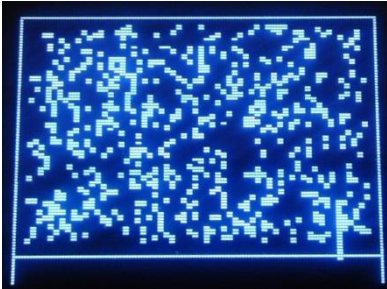
The second part of adding a maze to the game involved updating the collision and fruit generation checks. Previously, they were performed on a 2D grid that represented the display. Afterwards, the code was modified so as to check the linearized array that was given as an input to the display library. This update added the maze to the game.

Variable Speed

The speed of the game was varied by changing the SysTick Timer period setting. The timer is configured using a divider with respect to the oscillator frequency. Changing the divider changes the speed.

Screenshots of Gameplay

The following few figures show some of the pictures taken during actual gameplay and testing.

 <p>Score : 26</p>	 <p>Score : 8</p>	 <p>Score : 12</p>
<p>Figure 12 Screenshot of game without a maze. Borders only</p>	<p>Figure 13 Screenshot of game with a maze</p>	<p>Figure 14 Screenshot of game with another maze</p>
 <p>Right to cycle mazes 1 Left to cycle speeds 1 Up/Down for Vol U/D: 5 Press Select to Play</p>		
<p>Figure 15 Screenshot of Menu</p>	<p>Figure 16 Checking the random fruit generation</p>	

Concluding remarks

The snake game has been successfully implemented on the LM3S1968 Kit. It was initially decided to implement the game to a high level of detail and complexity (i.e. including mazes, sound, speed etc.). All the intended objectives have been met successfully. The only limitations faced were the license issues preventing full sound functionality.

During the course of the implementation, a lot of experience was gained, especially with respect to the design methodology and structuring the code in a modular way to allow further development. The game itself is simple. However, the implementation shows the breadth and complexity behind the front-end of even the simplest of games.

Appendix

The appendix gives the code for the entire implementation of the game. Before giving the code, it is necessary to explain what is in the various files.

- `main.c`
This contains the entry point for the game and global variables.
- `my_init.c`
This contains initializations for various peripherals and variables used in the game.
- `snake.c`
This file contains the main implementation of the game. The SysTick Interrupt routine is an excellent starting point to follow the code. The sub-functions called in the SysTick interrupt routine are contained within the file.

The function and variable naming used result in a self-explanatory code. Header files are also given in the appendix at the end. Files containing the data for the maze and the sounds are not included in the report. Also not included are the two libraries used (`class-d.c` for audio and `rit128x96x4.c` for display).

Main.c

```
#include <lm3s1968.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/lm3s1968.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/systick.h"
#include "driverlib/interrupt.h"
#include "rit128x96x4.h"
#include "my_init.h"
#include "my_display.h"
#include "snake.h"

// Global Variables
unsigned char Map_Display[Yres][Xres]; //2D map of the display
unsigned char Frame_Display[Xres*Yres/2]; //Linear array input required by Display Lib
struct Snake My_Snake[3072]; //Upper limit of snake length for a very pro player
int Current_Snake_Direction; //Current direction of the head of snake
int Snake_Tail_Index; //Tail index for location of tail in My_Snake
int FruitX; //X coordinate of Current fruit location
int FruitY; //Y coordinate of Current fruit location
int Pause; //Game in Pause/Unpause mode.
int Score = 0; //Score counter
int Speed = 1; //Speed of snake. Related to divider of Systick
int Volume = 5; //Volume of the sound
int Maze = 1; //Maze selection. 1,2,3 possible.
int Game_Mode = 0; //Game/Menu mode selector.

int main(void) {

    Clock_Init();
    PortG_Init();
    Sound_Init();
    OLED_Init();
    LED_Blink();
    SysTickInt_Init();
    IntMasterEnable();
    while(1) {
        // SLEEP
    }
}
```

My_init.c

```
//INIT FUNCTIONS

#include "my_init.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/lm3s1968.h"
#include "inc/hw_ints.h"
#include "inc/hw_sysctl.h"

#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/systick.h"
#include "rit128x96x4.h"

#include "class-d.h"
#include "snake.h"
#include "my_sounds.h"
//Init Clocks
```

```

//Init Int
//Init SysTick
//Init OLED
//Init Port G

void Clock_Init(void){

    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_8MHZ);
}

void PortG_Init(void){

    //Port Enable
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
    SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOG);

    //Gating
    SysCtlPeripheralClockGating(true);

    //LED
    GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);

    // Set pins 3, 4, 5, 6 as input
    GPIOPinTypeGPIOInput(GPIO_PORTG_BASE, GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6
| GPIO_PIN_7);
    GPIODirModeSet(GPIO_PORTG_BASE,GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 |
GPIO_PIN_7, GPIO_DIR_MODE_IN);

    //Pull-up
    GPIOPadConfigSet(GPIO_PORTG_BASE,
GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6|GPIO_PIN_7,
GPIO_STRENGTH_2MA,
GPIO_PIN_TYPE_STD_WPU);
}

void SysTickInt_Init(void) {

    SysTickPeriodSet(SysCtlClockGet()/5);
    SysTickIntEnable();
    SysTickEnable();
}

void GPIOInt_Init(void){

    GPIOIntTypeSet(GPIO_PORTG_BASE,BUTTON_UP|BUTTON_DOWN|BUTTON_LEFT|BUTTON_RIGHT|BUTTON_SE
LECT,GPIO_FALLING_EDGE);
    GPIOPinIntClear(GPIO_PORTG_BASE,BUTTON_UP|BUTTON_DOWN|BUTTON_LEFT|BUTTON_RIGHT|BUTTON_SE
LECT);
    GPIOPinIntEnable(GPIO_PORTG_BASE,BUTTON_UP|BUTTON_DOWN|BUTTON_LEFT|BUTTON_RIGHT|BUTTON_S
ELECT);
    IntPrioritySet(INT_GPIOG, 0x05);
    IntEnable(INT_GPIOG);
    //IntMasterEnable();
}

void OLED_Init(void){
    RIT128x96x4Init(1000000);
}

```



```

void Snake_Init(void){

    int n;
    int x_init = 50;
    Snake_Tail_Index = 20;
    Current_Snake_Direction = DIRECTION_RIGHT;
    for (n = 0 ; n < Snake_Tail_Index ; n++){

        My_Snake[n].Direction = DIRECTION_RIGHT;
        My_Snake[n].LocationX = x_init--;
        My_Snake[n].LocationY = 50;
    }

    //First rand would always show at a wierd place. So twice.
    FruitX = 75; //Init to avoid clearing corner of maze.
    FruitY = 75;
    Generate_Fruit();
    Generate_Fruit();
}

void Sound_Init(void){

    if(REVISION_IS_A2)
    {
        SysCtlLDOSet(SYSCTL_LDO_2_75V);
    }
    SysCtlPWMClockSet(SYSCTL_PWMDIV_1);
    ClassDInit(SysCtlClockGet());
    while(ClassDBusy())
    {
    }

    //    ClassDPlayADPCM(g_pucIntro, sizeof(g_pucIntro));
}

```

Snake.c

```
//SNAKE.C
//MAIN FUNCTIONS FOR SNAKE AND MOVEMENT HERE
#include "stdlib.h"
#include "stdio.h"
#include "my_init.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/lm3s1968.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "rit128x96x4.h"
#include "driverlib/systick.h"
#include "driverlib/interrupt.h"
#include "class-d.h"

#include "my_sounds.h"
#include "my_maze.h"
#include "snake.h"

//SYSTICK Interrupt

void SysTick_Handler(void) {

    if(Game_Mode)
    {
        //Systick Effect while gaming
        LED_Blink();
        Check_Collision();
        Check_Eatable();
        Update_Snake();
        Update_Display();
        //Generate_Fruit();
    }
    else
    {
        //Systick to read GPIO while in Menu
        Menu();
        ClassDPlayADPCM(bgMusic, lengthofbgMusic);
    }

}

// GPIO Interrupt Handler
void GPIOPortG_Handler(void){

    volatile unsigned long rawStatus = GPIOPinIntStatus(GPIO_PORTG_BASE,false);

    GPIOPinIntClear(GPIO_PORTG_BASE,BUTTON_UP|BUTTON_DOWN|BUTTON_LEFT|BUTTON_RIGHT|BUTTON_SELECT);

    if (rawStatus & BUTTON_UP) {
        if(Current_Snake_Direction != DIRECTION_DOWN)
            Current_Snake_Direction = DIRECTION_UP;
    }

    if (rawStatus & BUTTON_DOWN) {
        if(Current_Snake_Direction != DIRECTION_UP)
            Current_Snake_Direction = DIRECTION_DOWN;
    }

}
```

```

    if (rawStatus & BUTTON_LEFT) {
        if(Current_Snake_Direction != DIRECTION_RIGHT)
            Current_Snake_Direction = DIRECTION_LEFT;
    }

    if (rawStatus & BUTTON_RIGHT) {
        if(Current_Snake_Direction != DIRECTION_LEFT)
            Current_Snake_Direction = DIRECTION_RIGHT;
    }

    if (rawStatus & BUTTON_SELECT) {

        if(Pause){
            SysTickDisable();
            Pause = 0;
        }
        else {
            SysTickEnable();
            Pause = 1;
        }
    }
}

void Check_Collision(void){

    int Map_Collision_Check , Snake_Collision_Check , n;
    Snake_Collision_Check = 0;

    Map_Collision_Check = Frame_Display[My_Snake[0].LocationY*64
+My_Snake[0].LocationX/2]
        + Frame_Display[(My_Snake[0].LocationY+1)*64 + My_Snake[0].LocationX/2]
        + Frame_Display[My_Snake[0].LocationY*64 + (My_Snake[0].LocationX+1)/2]
        + Frame_Display[(My_Snake[0].LocationY-1)*64 +My_Snake[0].LocationX/2]
        + Frame_Display[My_Snake[0].LocationY*64 + (My_Snake[0].LocationX-1)/2]
        + Frame_Display[(My_Snake[0].LocationY+1)*64 + (My_Snake[0].LocationX+1)/2]
        + Frame_Display[(My_Snake[0].LocationY-1)*64 + (My_Snake[0].LocationX-
1)/2];

    for (n = 1 ; n < Snake_Tail_Index ; n++){
        if( (My_Snake[0].LocationY == My_Snake[n].LocationY) &&
(My_Snake[0].LocationX == My_Snake[n].LocationX))
            Snake_Collision_Check = 1;
    }

    if( Map_Collision_Check || Snake_Collision_Check )
        SysTickDisable();
}

void Check_Eatable(void){
    //Check if snake head ate the fruit
    if(
        ((My_Snake[0].LocationY == FruitY) || (My_Snake[0].LocationY + 1 == FruitY) ||
(My_Snake[0].LocationY - 1 == FruitY)) &&
        ((My_Snake[0].LocationX == FruitX) || (My_Snake[0].LocationX + 1 == FruitX) ||
(My_Snake[0].LocationX - 1 == FruitX))
    )
    {
        Score = Score + Speed;
        Update_Score();

        Generate_Fruit();
    }
}

```

```

        My_Snake[Snake_Tail_Index].Direction = My_Snake[Snake_Tail_Index-1].Direction;
        My_Snake[Snake_Tail_Index].LocationY = My_Snake[Snake_Tail_Index-1].LocationY;

        My_Snake[Snake_Tail_Index].LocationX = My_Snake[Snake_Tail_Index-1].LocationX;

        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_UP)
            My_Snake[Snake_Tail_Index].LocationY = My_Snake[Snake_Tail_Index-
1].LocationY + 1;
        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_DOWN)
            My_Snake[Snake_Tail_Index].LocationY = My_Snake[Snake_Tail_Index-
1].LocationY - 1;
        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_LEFT)
            My_Snake[Snake_Tail_Index].LocationX = My_Snake[Snake_Tail_Index-
1].LocationX + 1;
        if(My_Snake[Snake_Tail_Index].Direction == DIRECTION_RIGHT)
            My_Snake[Snake_Tail_Index].LocationX = My_Snake[Snake_Tail_Index-
1].LocationX - 1;
        Snake_Tail_Index = Snake_Tail_Index + 1;

        ClassDStop();
        ClassDPlayADPCM(eat_sound, lengthofeat_sound);
    }
}

void Update_Snake(void){

    int n;

    //Delete Tail of snake
    const unsigned char AntiBlob[] = {0x00 , 0x00, 0x00, 0x00};
    RIT128x96x4ImageDraw(AntiBlob,My_Snake[Snake_Tail_Index-
1].LocationX,My_Snake[Snake_Tail_Index-1].LocationY,2,2);

    //Loop to update snake structures current position and next directions
    for (n = 0 ; n < Snake_Tail_Index; n++){

        if(My_Snake[n].Direction == DIRECTION_UP)
            My_Snake[n].LocationY = My_Snake[n].LocationY - 1;
        if(My_Snake[n].Direction == DIRECTION_DOWN)
            My_Snake[n].LocationY = My_Snake[n].LocationY + 1;
        if(My_Snake[n].Direction == DIRECTION_LEFT)
            My_Snake[n].LocationX = My_Snake[n].LocationX - 1;
        if(My_Snake[n].Direction == DIRECTION_RIGHT)
            My_Snake[n].LocationX = My_Snake[n].LocationX + 1;

    }

    for (n=Snake_Tail_Index ; n > 0; n--){

        My_Snake[n].Direction = My_Snake[n-1].Direction;
    }

    My_Snake[0].Direction = Current_Snake_Direction;

}

void Generate_Fruit(void){

    int randX,randY,check,i;
    const unsigned char Fruit[] = {0xFF , 0xFF, 0xFF, 0xFF};
    const unsigned char AntiFruit[] = {0x00 , 0x00, 0x00, 0x00};
    check = 1;

```

```

do{
    randX = 5 + rand() % ( Xres-10 );
    randY = 3 + rand() % ( Yres-18 );

    if(
        (Frame_Display[(randY * 64) + (randX / 2)] > 0)||
        (Frame_Display[((randY+1) * 64) + (randX / 2)] > 0)||
        (Frame_Display[(randY * 64) + ((randX+1) / 2)] > 0)||
        (Frame_Display[((randY-1) * 64) + (randX / 2)] > 0)||
        (Frame_Display[((randY+1) * 64) + ((randX+1) / 2)] > 0)||
        (Frame_Display[((randY-1)* 64) + ((randX-1) / 2)] > 0)
    )
    {
        for(i = 0 ; i< Snake_Tail_Index ; i++)
        {
            if(
                (My_Snake[i].LocationX - randX > -2 && My_Snake[i].LocationY - randY > -2) &&
                (My_Snake[i].LocationX - randX < +2 && My_Snake[i].LocationY - randY < +2)
            )
            {
                check = 1;
            }
        }
    }
    else
        check = 0;
}while(check);

    RIT128x96x4ImageDraw(AntiFruit,FruitX,FruitY,2,2);
    FruitX = randX;
    FruitY = randY;
    RIT128x96x4ImageDraw(Fruit,FruitX,FruitY,2,2);
}

void Update_Display(void){
    //Update the Display to the new head and tail of the snake.
    int n;
    const unsigned char Blob[] = {0xFF , 0xFF, 0xFF, 0xFF};

    for (n = 0 ; n < Snake_Tail_Index; n++){
        RIT128x96x4ImageDraw(Blob,My_Snake[n].LocationX,My_Snake[n].LocationY,2,2);
    }
}

//Switch ON/Off LED functions here
void LED_On(void){
    GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0xFF);
}

void LED_Off(void){
    GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x00);
}

```

```

}

void LED_Blink(void){

    LED_On();
    SysCtlDelay(100000);
    LED_Off();

}

void Diplay_Maze_Init(void){

    int x;
    int y;
    int i;

    //CLEAR DISPLAY
    for(y = 0; y < Yres ; y++){

        for(x = 0; x < Xres; x++){

            Map_Display[y][x] = 0x00;

        }

    }

    switch(Maze)
    {

    case 1:

        //MAKE BORDER
        for(y = 0; y < Yres ; y++){
            Map_Display[y][0] = 0xFF;
            Map_Display[y][Xres-1] = 0xFF;
        }

        for(x = 0; x < Xres ; x++){
            Map_Display[0][x] = 0xFF;
            Map_Display[Yres-10][x] = 0xFF;
        }

        //
        // Loop over the 94 scan lines of the display.
        //
        for(y = 0; y < Yres; y++)
        {
            //
            // Loop over the 128 columns of the display.
            //
            for(x = 0; x < Xres ; x += 2)
            {
                //
                // Display these two columns of the maze.
                //
                Frame_Display[(y * 64) + (x / 2)] = ((Map_Display[y][x] ? 0xf0 :
0x00) |
                (Map_Display[y][x + 1] ? 0x0f : 0x00));
            }

        }

        break;
    }
}

```

```

case 2:
    //Load from header
    for(i = 0; i < Yres*Xres/2 ; i++){
        Frame_Display[i] = Maze_2[i];
    }
    break;

case 3:
    //Load from header
    for(i = 0; i < Yres*Xres/2 ; i++){
        Frame_Display[i] = Maze_3[i];
    }
    break;

default:
    //MAKE BORDER
    for(y = 0; y < Yres ; y++){
        Map_Display[y][0] = 0xFF;
        Map_Display[y][Xres-1] = 0xFF;
    }

    for(x = 0; x < Xres ; x++){
        Map_Display[0][x] = 0xFF;
        Map_Display[Yres-10][x] = 0xFF;
    }

    //
    // Loop over the 94 scan lines of the display.
    //
    for(y = 0; y < Yres; y++)
    {
        //
        // Loop over the 128 columns of the display.
        //
        for(x = 0; x < Xres ; x += 2)
        {
            //
            // Display these two columns of the maze.
            //
            Frame_Display[(y * 64) + (x / 2)] = ((Map_Display[y][x] ? 0xf0 :
0x00) |
            (Map_Display[y][x + 1] ? 0x0f : 0x00));
        }
        break;
    }

    RIT128x96x4ImageDraw(Frame_Display,0,0,Xres,Yres);
}

void Update_Score(void){
    char buffer[20];
    RIT128x96x4StringDraw("Score :",5,Yres-8,15);
    sprintf(buffer, "%d",Score);
    RIT128x96x4StringDraw(buffer,50,Yres-8,15);
}

void Menu(){
    char buffer[20];

    RIT128x96x4StringDraw("Press Select to Play",5,Yres-18,15);

```



```

RIT128x96x4StringDraw("Up/Down for Vol U/D:",5,Yres-38,15);
sprintf(buffer, "%d",Volume);
RIT128x96x4StringDraw(buffer,100,Yres-28,15);

RIT128x96x4StringDraw("Left to cycle speeds",5,Yres-58,15);
sprintf(buffer, "%d",Speed);
RIT128x96x4StringDraw(buffer,100,Yres-48,15);

RIT128x96x4StringDraw("Right to cycle mazes",5,Yres-78,15);
sprintf(buffer, "%d",Maze);
RIT128x96x4StringDraw(buffer,100,Yres-68,15);

if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_UP)) == 0) {
    Volume = Volume + 1;
    if (Volume > 10)
        Volume = 10;

    ClassDVolumeSet(Volume * 20);
}

if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_DOWN)) == 0) {
    Volume = Volume - 1;
    if(Volume < 0)
        Volume = 0;

    ClassDVolumeSet(Volume * 20);
}

if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_LEFT)) == 0) {
    Speed = Speed + 1;
    if (Speed > 5)
        Speed = 1;
}

if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_RIGHT)) == 0) {
    Maze = Maze + 1;
    if(Maze > 3)
        Maze = 1;
}

if (GPIOPinRead(GPIO_PORTG_BASE,(BUTTON_SELECT)) == 0) {
    SysTickIntDisable();
    SysTickDisable();
    Game_Mode = 1;
}

if(Game_Mode == 1)
{
    Diplay_Maze_Init();
    Snake_Init();
    GPIOInt_Init();
    SysTickPeriodSet(SysCtlClockGet()/(Speed*20));
    SysTickEnable();
    SysTickIntEnable();
}
}

```

My_init.h

```
#ifndef __MY_INIT_H__
#define __MY_INIT_H__

void Clock_Init(void);
void PortG_Init(void);
void SysTickInt_Init(void);
void GPIOInt_Init(void);
void OLED_Init(void);
void Snake_Init(void);
void Sound_Init(void);
#endif // __MY_INIT_H__
```

Snake.h

```
#ifndef __SNAKE_H__
#define __SNAKE_H__

//Defines
//Direction
#define DIRECTION_UP 1
#define DIRECTION_DOWN 2
#define DIRECTION_LEFT 3
#define DIRECTION_RIGHT 4
//Buttons
#define BUTTON_UP GPIO_PIN_3
#define BUTTON_DOWN GPIO_PIN_4
#define BUTTON_LEFT GPIO_PIN_5
#define BUTTON_RIGHT GPIO_PIN_6
#define BUTTON_SELECT GPIO_PIN_7

//Display
#define Xres 128
#define Yres 96
// Global Variables
extern unsigned char Map_Display[Yres][Xres];
extern unsigned char Frame_Display[Xres*Yres/2];
extern struct Snake My_Snake[3072]; //Upper limit of snake length for a very pro player :p
extern int Current_Snake_Direction;
extern int Snake_Head_Index;
extern int Snake_Tail_Index;
extern int FruitX;
extern int FruitY;
extern int Pause;
extern int Score;
extern int Speed;
extern int Maze;
extern int Volume;
extern int Game_Mode;

struct Snake
{
    int Direction;
    int LocationX;
    int LocationY;
};

// Function Prototypes
void GPIOPortG_Handler(void);
```

```
void Check_Collision(void);
void Check_Eatable(void);
void Update_Snake(void);
void Generate_Fruit(void);
void Update_Display(void);
void LED_On(void);
void LED_Off(void);
void LED_Blink(void);
void SysTick_Handler(void);
void Display_Maze_Init(void);
void Update_Score(void);
void Menu(void);

#endif // __SNAKE_H__
```

References

- ⁱ http://en.wikipedia.org/wiki/Nokia_3310
- ⁱⁱ <http://www.ti.com/lit/ug/spmu037a/spmu037a.pdf>
- ⁱⁱⁱ http://en.wikipedia.org/wiki/Snake_%28video_game%29
- ^{iv} <http://img.gfx.no/486/486041/3310-08.jpg>
- ^v <http://3.bp.blogspot.com/-ir45rXBfumI/T3B3AdEdsaI/AAAAAAAIIdI/4wfEadykI94/s1600/thosedays+%25281%2529.jpg>
- ^{vi} <http://audacity.sourceforge.net/>
- ^{vii} http://e2e.ti.com/support/microcontrollers/stellaris_arm_cortex-m3_microcontroller/f/473/t/46253.aspx