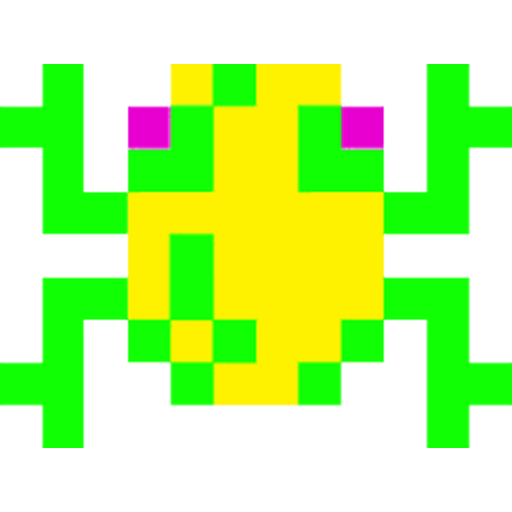
**FROGGER**

**by Evan Brown and Jorge Xelo**

epbrown jorgexel



**Table of Contents:**

*Division of work………………………………………………….………………………………………………………………………3*

*Objective……………………………………………………………………………………………………………………………………3*

*Development…..…………………………………………………………………………………………………………………………3*

*How to use…………………………………………………………………………………………………………………………………3*

*Main Program Flowchart……………………………………………………………………………………………………………4*

*Subroutine Flowcharts..……………………………………………………………………………………………………………..5*

*Timer0Handler...……….……………………………………………………..………….…………………………………5*

*Timer1Handler..……….………………………………………………………………….…………………………………6*

*UartHandler ……..…….…….…………………..………………………………………………………….………………7*

*PortAHandler.….………………….….………………………………………………………………………………………8*

*mov\_left……………...….…………….……………………………………………………………………………………………………9*

*Variable Offsets.………………………………………………………………………………………………………………………10*

**Division of Work:**

epbrown – main program, documentation

jorgexel – lab support

**Objective:**

The purpose of this program is to re-create the classic arcade game “Frogger” using PuTTy and a Tiva C ARM processor. The game is ascii character based, using different letters to represent the objects on the screen. The game attempts to get as close to the original as possible, while maintaining a relatively simple design and development process. Part of the game’s mechanics are created using different peripherals such as a keypad and LEDs on a GPIO board. The RGB LED on the Tiva C is also utilized to present different game states. Overall, the goal was to test the knowledge we had gained by using all of the different peripherals we had used throughout the semester.

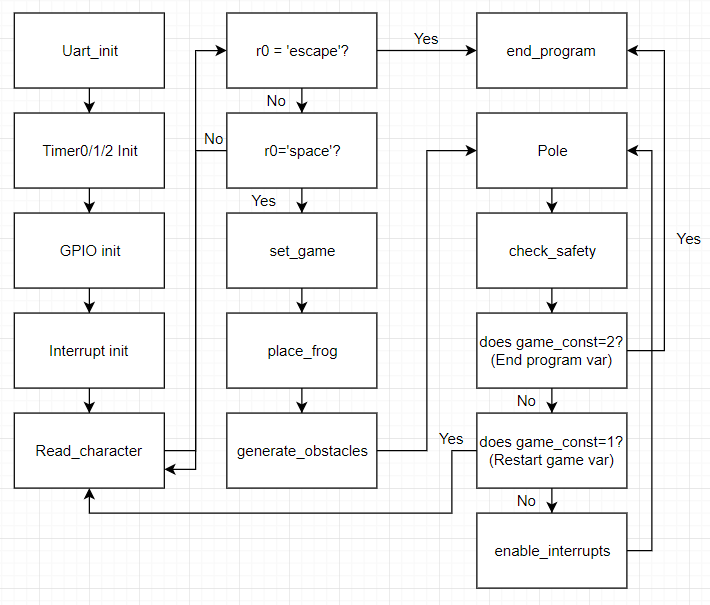
**Development:**

Much of the development of the program was spent on string manipulation and creating pseudo random numbers to make a random looking game board. Naturally this is where most of the debugging happened, as many problems arose with generating objects and moving them across the screen. Most issues that came up were simple fixes however, and could be solved by either looking at memory, or just stepping through the logic of the program. Adding all of the different functionalities of the program, such as lives, time limit, LEDs, etc., were quite trivial and required little debugging. Most of these functions were implemented using dedicated spots in memory that held all of the necessary values such as lives. Lastly, the entire program was written in Assembly except for a small C function that returned the mod of two numbers, due to the div\_and\_mod function in the library not working correctly with the relatively large numbers being used.

**How to use:**

To play the game there are instructions presented at the start, explaining how the frog moves, how points are earned, how to get to the next level, etc. It is close to the original game, where you have to get the frog across the road and the water without hitting the vehicles or falling in the water to get the frog home. The player has 4 lives and a 60 second time limit to get two frogs home. When the frogs are home, the level increases where the speed slightly increases and the time limit decreases. The experience is further improved through the use of the LEDs to show your lives, an RGB LED to show the status of the game, and a keypad to pause or resume the game.

**Main Program Flowchart:**

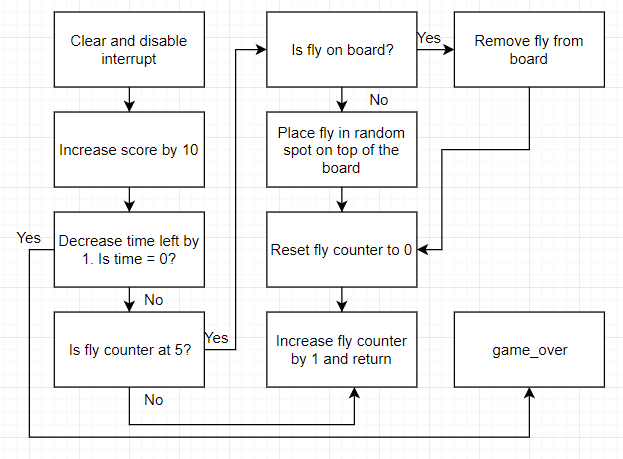


Summary:

After all initialization, the program poles read\_character until the user starts the game. When startup is initiated, set\_game and place\_frog are called, which sets all necessary variables to their initial values (such as #lives, #time, etc.) and places the frog randomly at the bottom of the board. Generate\_obstacles calls the function generate\_and\_move about 50 times. Generate\_and\_move is called during Timer1Handler as well and moves all lanes by one position accordingly while also randomly generating new objects. As you could imagine this then imitates an instant generation of objects, as the board will be filled up with objects since its width is less than 50. Then, the program poles check\_safety until the program is either restarted or terminated. Check\_safety constantly checks what char the frog is sitting on and takes appropriate action. If the char is not safe, the frog is killed, if it is a home spot, the # of makes is increased, and if it’s a fly, 100 points are rewarded.

**Subroutine Flowcharts:**

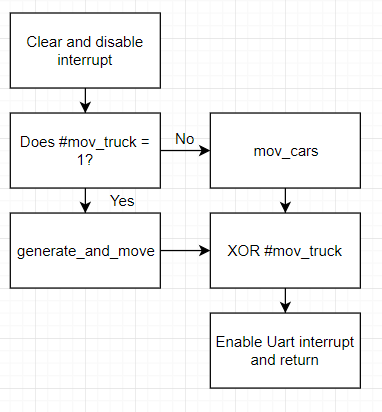
Timer0Handler:



Summary:

The Timer0Handler fires once every second. The time variable is decreased by 1 every time and if it reaches zero the game is lost. Game\_over sets the RGB to blue to indicate game over, then disables all interrupts and prints a game over screen, which then prompts the user to restart or end the program. The fly counter is then checked to determine if a fly should be added or removed. It is decided that five seconds is the time between fly generation. When the fly is done being added or removed, the counter is reset to zero. The counter is increased by 1 and returns.

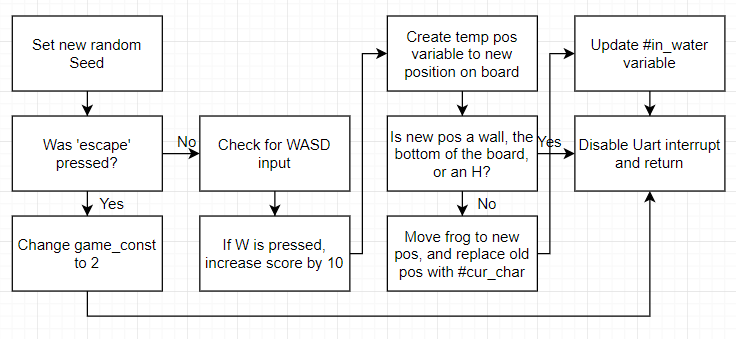
Timer1Handler:



Summary:

All the Timer1Handler does is move objects on the board. It fires every half second (on level 1), and checks the mov\_trk variable to see if everything should be moved or just the cars, since the cars move every half second while the rest move every second. generate\_and\_move moves everything on the board by one, while mov\_cars only moves the cars by one position. mov\_truck is then XORed with itself to reverse the value for the next call. Lastly, the Uart interrupt is enabled so the frog can move again (since the frog can only move as fast as the cars).

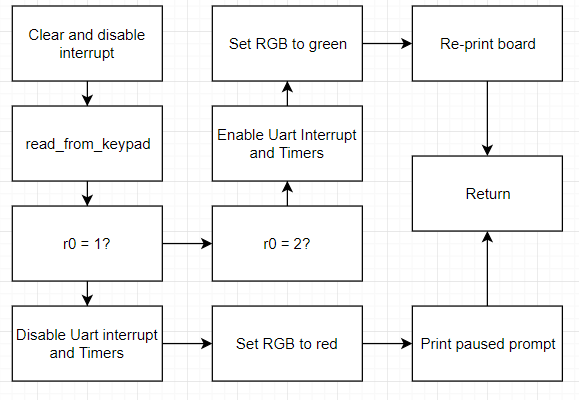
Uart0Handler:



Summary:

Calls function set\_seed which takes the value from Timer2 to create a new random sequence. This is stretched out across an array and uses the other Timers to create pseudo random numbers from the randomness of user input. The handler then checks for an escape press or WASD input, and either sets the program to end or moves the position accordingly. The new temp position then checks if there is a border or a home spot there and only moves if it is a valid spot. The in\_water variable is then updated accordingly and the interrupt is disabled until the Timer1Handler is fired.

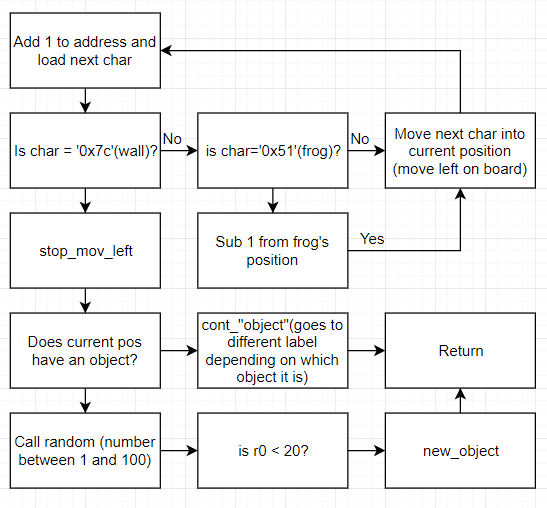
PortAHandler:



Summary:

Takes in keypad input and checks if it is to be paused or resumed. A ‘1’ indicates a pause and a ‘2’ indicates a resume. The RGB LED is lit accordingly when paused or resumed to indicate game state. When paused, a prompt is printed saying the game is paused and to press ‘2’ to resume.

mov\_left:



Summary:

This and mov\_right (the same but add is sub and vice versa) are called for every lane of the board in the function generate\_and\_move. Every character is checked and moved until the wall is reached. When the end is reached, if there is an object currently on the edge of the board, that object will continue to generate until the proper length is reached (this is checked through a loop that goes the opposite direction on the board to check how long the object currently is, and then either adds to it or leaves it). If there is no object there, a random number is called to determine if one is to be generated. If it is less than 20, a random character is placed there depending on if it’s in water or on land to be generated further on the next call to mov\_left.

**Variable offsets:**

The following are offsets for the base address in memory position 0x20000450, used to hold variables for the program. Some of these appear in the flowcharts.

#cur\_char: Stores the character the frog is sitting on currently

#in\_water: Set to 1 if frog is on the river, 0 if on land

#lives: Holds the current amount of lives

#makes: Holds the current amount of frogs at home

#score: Holds the current score

#level: Holds the current level

#time: Holds current time left

#mov\_trk: Set to 1 if whole board should move, 0 if only the cars should move

#game\_const: Set to 2 if program is to end, 1 if game is over, 0 if game is running

#fly: Fly counter that counts up to 5 to determine if fly is to be placed or removed

#fly\_in: Set to 1 if fly is on board, 0 if not

#rand\_const: Contains the timer value when the user last moved

#seed\_arr\_pos: Position in seed array

#seed\_array: Contains several seeds for the random function

#temp\_string: Used for strings during int conversion