CSE 379
Lab 7: Space Invaders

University at Buffalo

Hang Lin (hlin42)
Jay J Chen (jchen247)

May 7, 2018

Contents

1 Description / Overview

In our last and final lab, we are met with a difficult task. And, that task is to put all our knowledge of our previous labs, which included:

1. *div_and_mod* lab
2. *GPIO* lab
3. *interrupts* lab
4. *strobing* lab

We are to write in ARM assembly language / C language a game called *Micro Space Invaders*. This is the ultimate test of skill and knowledge of ARM ASM language. To make this even more challenging for us, we wrote the ENTIRE game in ARM assembly language.

1.1 Synopsis and Objective

The version of *Micro Space Invaders* we are asked to code is rather a simplified version. In other words, the 1978 version where computers were only at their third generation, and where Intel has introduced their first 16 bit 8086 chip. Our version of the game should include the same function as the original: shoot down enemy invaders to before it reaches your ship to earn points and to advance to the next level. Our ship with start with 4 lives. and the invaders will start at the same position every level. The invaders can only move at a set speed every level, and everytime they reach the wall, their movement will go the other direction. This version of the game will also incorporate strong shields (S) and weak shields (s). When strong shield is shot by any projectile, it will become a weak one, then a weak one will become nothing. We are given 120 seconds to complete the game, and a scoreboard will print promptly after ending the game.

1.2 Usage, Layout, and Basic Functionalities

Before running PuTTy, the user must find the COM port the UART board is attached to. To do this, fire up command prompt (CMD) and run the command *mode*. The last shown COM port is usually the UART COM port. Then, on uVision, in the *Options for target* window, the Xtal speed must be set to 14.7456 MHz. And on the *Linker* tab, select *User memory layout*. Also, in the *Debug* tab, select *ULINK ARM Debugger*. Now, build the program and flash/download onto the UART board. Run PuTTy and set "Serial" as connection type, and type in the corresponding COM port found earlier. For the baud rate, specifically our baud rate, we used 1152000. To figure this out, refer to the documentations. A blank terminal will appear. Hit the reset button on the UART board and follow the instructions printed. Instructions are self explanatory.

This is the basic layout of the board:

```
|--------------------|
|empty for space ship|
|       OOOOOOO      |
|       MMMMMMM      |
|       MMMMMMM      |
|       WWWWWWW      |
|       WWWWWWW      |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|   SSS   SSS   SSS  |
|   S S   S S   S S  |
|                    |
|          A         |
|--------------------|
```

Legend

```
|,-: Wall
W:   Invader (10 pts)
M:   Invader (20 pts)
O:   Invader (40 pts)
A:   Player
S:   Strong Shield
S:   Weak Shield
X:   Bonus Motherships
^:   Player's Shots
V:   Enemy's Shots
```

1.2.1 Board

Everything must be contained and operated within the board. Excluding the walls, the playable area is 21 by 15. As seen by the layout before, there are walls surrounding the perimeter, and within the walls there are other pieces of the game. Every next line is hex 19. For example, the space directly below any invader have 25 spaces, or hex 19. The board should not be destroyed at any given time, nor anything should move out of it.

1.2.2 Enemy invaders

They consist of 14 W enemies, each worth 10 points, 14 M enemies, each worth 20 points, and 7 W enemies, worth 40 points; a total of 700 points. Their movement is simple every level, with the only difference being speed. Speed will increase by a certain amount every next level. The enemies will move one space to the left or the right, until it hits a wall. When it hits a wall, it will shift down one space then, continue moving the other way.

1.2.3 Mothership

Represented by a 'X'. This score is randomly generated everytime you hit it. A real time feedback will be displayed somewhere on the screen. The mothership can only move in one direction everytime it appears, and can only appear in the 'empty for space ship' location; which is the top most row. The direction of the movement, left to right or right to left, is also randomly generated. It will disappear upon reaching the wall. Additionally, the speed of the mothership is faster than the movements of the invaders. Only one mothership can be on the board at any given time.

1.2.4 Player Ship

The player ship is denoted by 'A', and can move left and right within the bottom most row. It cannot move out of the wall. The player movement is all done within the UART interrupt handler since using read character would be problematic later on. The player ship will also only contain 4 lives. The player loses one every time she or he is shot by the enemy invaders projectile.

1.2.5 Enemy Projectile

The enemy projectile is denoted by a 'v'. It will be randomly shot by any invaders that is still alive. It will advance in sync with our timer speed. And, it can only move forward. There are several of cases when it hits something. If it hits a strong shield, 'S', it will become a weak shield, 's". A weak shield will become nothing. If it hits the player ship, 'A', the player will lose one life and 100 points. And lastly, if it hits the wall, the bullet will disappear. Only one enemy projectile can be on the board at any given time.

1.2.6 Player Ship Projectile

The player ship projectile is denoted by a '^'. It will be shot whenever the player command it to. To eject a projectile, the player must hit the spacebar key, and this projectile can only move forward. There are several of cases when it hits something. If it strucka strong shield, 'S', it will become a weak shield, 's". A weak shield will become nothing. If it hits an enemy invader, the enemy invader will disappear with the projectile, and the player is rewarded points corresponding to the enemy. The mothership can be hit, but extremely hard, but if the player hit it, they will be granted extra points. If the projectile hits the wall, nothing will happen. It will simply disappear. Only one player projectile can be on the board at any given time.

1.2.7 Shields

The shields are denoted by 'S' for strong shield and 's' for weak shield. When the strong shield is hit by any projectile(s), it will become a weak shield. And, if that is hit, it will become nothing. The player is granted 15 strong shield in the beginning of every level, and the shield are stationary and are unable to move.

1.3 Other Functionalities

Other miscellaneous functionalities, such as the GPIO on the UART board, and other necessary functions to run the game.

1.3.1 New Level

The current level will be displayed right underneath the board. Originally, the game will start in level 0, and will increase everytime the player finishes killing all invaders on that level. Then it will reach the next level. An entirely new board will appear up. The player will have to destroy all enemy once again. But the difference on every level is that the enemy will move faster than the previous level. The current score will be reset, but the total score will still be stored somewhere to be displayed later on. The amount of lives will remain the same.

1.3.2 Seven Segment Display

The current score will be displayed on the Seven Segment. To get this working with only 2 timers that were given, we wrote another routine that given us another timer essentially. In the end, it is just a software timer.

1.3.3 RGB LED

This will display the current game status. It is blue if the game is paused. Green when the game is in progress, flashing red when player shoots a bullet or get shot, and purple when the game is over. The flashing red function was done after the submission. To get this working we decided to add an infinite loop everytime the enemy has struck player ship. It will flash red in that loop, and the user will be shown, 'You have died, press Enter to continue.' We will call the transfer ready register to see if the Enter key was depressed. Then it will jump out of the loop and continue on regular gameplay.

1.3.4 LEDs

These LED functions as lives, and are the only way to see how many lives the player has left. The game starts with 4 lives, and the player will lose one life everytime he or she is hit by an emeny projectile. When all lives are gone, game is over.

1.3.5 Momentary Push Button P0.14

This button acts as a pause and un-pause function. When paused everything is in stasis in the game. Player will not be able to move, and all the timers will come to a stand still. In general, it nothing will move or respond. When the button is pressed again, timers will come back up again, as well as player inputs.

1.3.6 Scoring

The player will be able to see their current level score on the seven segment. The total score is stored in a memory location, and will constantly be updated. It will add the current score every time, and store it into a memory. Total score will be displayed later on.

1.3.7 Timers

We were only given two timers to work with. One timer we used for enemy motion, and the other was for projectile speed as well as mothership speed. We also needed one for the strobing the points on the seven segment display. To do this, the programmer have to implement their own timer, other words a software timer. We also had to implement a 2 minute time for the game. When the 2 minutes is up, the game ends.

1.3.8 Finishing the Game

There are several of ways the player can lose the game. One, is when all 4 lives are depleted. Two, is when the enemy reaches the shield. Third, when the timer hits 120 seconds. And lastly, the user can just exit the game. Upon reaching the end of the game, a detailed report or scoreboard will be displayed for the player. The user will see how many lives they lost on each level, and time used on each level, as well as the points they earned on each level. Total time will be displayed in the overview section as well as the total score.

The user will be prompted if they want to restart the game. That is done by entering the 'Enter" key. Or the user can simply hit the red 'X' button on the window to exit. This was done later after the submission date. But the logic behind this was simply in our end game, we added an infinite loop that will keep checking the transfer ready register to see if the Enter key was pressed, then it will branch to the beginning of the entire program. As simple as that.

1.4 Memory Addresses Used

In this lab we used many memory addresses to store our setups. For example, the amount of lives, we will store a 4 in its corresponding memory address, or a 1 in our pause game flag memory address. Here is a table of the setup for each memory space we used to store data.

Name	Memory Address	Initial Value Stored	Description
total_score_1	0x40006920	0	Ones place for our total score
total_score_10	0x40006924	0	Tens place for our total score
total_score_100	0x40006928	0	Hundreds place for our total score
total_score_1000	0x4000692C	0	Thousands place for our total score
current_score_1	0x40006900	0	Ones place for our current score
current_score_10	0x40006904	0	Tens place for our current score
current_score_100	0x40006908	0	Hundreds place for our current score
current_score_1000	0x4000690C	0	Thousands place for our current score
lives	0x40005000	4	Player starts with 4 lives
time_left_1	0x40006944	0	Ones place for game timer
time_left_10	0x40006948	0	Tens place for game timer
time_left_100	0x4000694C	0	Hundreds place for game timer
time_left_total	0x40006954	0	Check if total time hits 120 seconds
strobing_flag	0x40005040	1	To strobe current score onto 7-seg
promptoffset	0x40007100	N/A	Stores the address of game board
prompt_storage_offset	0x40007E20	N/A	Stores level breakdown
prompt_storage_offset_copy	0x40007E24	N/A	Read level breakdown and print
LED_1_command	0x40007D00	N/A	Stores digits for 7-seg to be displayed
seconds_increment_counter	0x40007AA0	0	Counts time forward
timer_counter	0x40007B00	0	Used as software timer for our strobing
keystork_rng_counter	0x40006940	48	Used for randomness
current_keyboard_rng_input	0x40007994	48	Used for randomness
promptend_offset	0x40007A00	N/A	Address of scoreboard stored
mothership_score_1	0x40006930	0	Ones place for mothership score
mothership_score_10	0x40006934	0	Tens place for mothership score
mothership_score_100	0x40006938	0	Hundreds place for mothership score
mothership_appears_count	0x40007E28	0	Amount of times mothership appears
mothership_appears_hit	0x40007E2C	0	Amount of times mothership was hit
paused_game_flag	0x40005010	1	1 for pause, 0 for un-pause
current_level	0x40000200	0	Determines player current level
mothership_exist_on_board	0x400078A0	0	Only one mothership is allowed
mothership_left_or_right_dir	0x40007860	0	0 for right 1 for left
endpoint_location	0x40006940	N/A	The offset where the first enemy hits middle
ship_location	0x40005000	N/A	The offset of board + 0x183 is initial location
enemyoffset	0x40007120	N/A	Topmost row of enemy offset. 0x19
enemy2offset	0x40007130	N/A	2nd topmost row enemy offset. Add 0x19
enemy3offset	0x40007140	N/A	Middle row enemy offset. Add 0x19
enemy4offset	0x40007150	N/A	4th row enemy offset. Add 0x19
enemy5offset	0x40007160	N/A	Last row enemy offset. Add 0x19
enemycounts	0x40007270	7	Topmost row enemy count
enemy2counts	0x40007280	7	2nd topmost row enemy count
enemy3counts	0x40007290	7	Middle row enemy count
enemy4counts	0x400072A0	7	4th row enemy count
enemy5counts	0x400072B0	7	Last row enemy count
right_enemycounts	0x40007370	7	Top row enemy count from right. Prevent bug
right_enemy2counts	0x40007380	7	2nd row enemy count from right. Prevent bug
right_enemy3counts	0x40007390	7	3rd row enemy count from right. Prevent bug
right_enemy4counts	0x400073A0	7	4th row enemy count from right. Prevent bug
right_enemy5counts	0x400073B0	7	Last row enemy count from right. Prevent bug

offset_hit_counts	0x40007500	0	Top row enemy struck, +1 to offset
offset2_hit_counts	0x40007520	0	2nd row enemy struck, +1 to offset
offset3_hit_counts	0x40007520	0	Middle row enemy struck, +1 to offset
offset4_hit_counts	0x40007500	0	4th row enemy struck, +1 to offset
offset5_hit_counts	0x40007500	0	Last row enemy struck, +1 to offset
left_right_dir_flag	0x400072F0	0	See where enemy will move when hit wall
flagof_just_reached_left_wall	0x4000700F	0	See if hit wall. If yes shift down
mothership_offset	0x40007880	N/A	Depend on RNG, left offset or right offset used
enemy_proj_offset	0x400074A0	N/A	Random generated. Look for enemy and shoot
enemy_proj_on_board	0x400074B0	0	One projectile at a time.
number_of_loops	0x40007544	0	Used for RNG calculation
proj_offset	0x40007400	N/A	Player ship projectile. Constant sub 0x19
ship_location	0x40006000	N/A	Current play location
proj_exist_on_board_flag	0x40007410	0	One projectile at a time
selected_enemy_that_shoot_proj	0x40007440	0	RNG for which will shoot projectile

Most of the setups are shown here on this table. These setups are all called in the beginning of the lab. The pause flag is the only initially set a 1 for pause. Instructions needs to be printed and shown to the player. The lives is set to 4 since we start with 4 lives and lose one everytime the player dies. Most things are set at 0, because that is how most thing begin. The RNG are initially 48 because we are adding 48 whenever user presses something. All enemy counts are at 7, because in every row there are 7 enemies. The rest are N/A because They all rely on the offsets of the gameboard or dependent on another thing.

1.5 How To Play

The instructions will print out on the PuTTy screen. It is recommended to keeps the cap locks buttons on; since the only keys the player needs is 'A', 'D' and spacebar. 'A' will be used to move left and 'D' will be used to move right. Spacebar will be used to fire projectiles to destroy enemy invaders. The 'Enter' key is used to restart the game upon reaching the game over screen. The P0.14 interrupt button is used to start the game, pause the game, or to resume the game.

1.6 Debugging

Before starting debugging session, in order to successfully debug without read/write memory errors, be sure to setup uVision to use the memory layout from target dialogue. To do this, goto the *Project* then into *Options for Target*. A dialogue box will come up, right there select the *Linker* tab and selecting *Use Memory Layout from Target Dialogue*, then hit accept. This way, there will be no errors popping up in the console during debugging. And also, in addition to doing this, make sure in the *Debug* tab, select *ULINK ARM Debugger*.

In this specific lab, we encounter many problems. We were thrown into the exception handler many many times. Most of the time it would bring us to a Data Abort and other times it would be Software Interrupt and Undefined Handler. It was an easy fix. Some of these fixes were using the wrong instructions. Such as instead of using LDRB we used LDR and vice versa. Same with STRB and STR. We made the changes promptly, and most of the problems were gone. Some still persists. For some of these fixes, we use Keil command, LTORG. This commands shuffle the literal pool. From what we know, it shuffles around the register and memory so our program doesn't get thrown into the exception handler.

When using LTORG some problems still persists, so we sprinkled LTORG here and there. Then we figured out, we are using too many branch and linkings (BL). We found out using too many of them will eventually cause problems with the link register. So we put some of the routine into their respective places instead of calling BL. And, that fixed all our problems. We did not use any LTORGs and we were not thrown into the handler anymore.

1.7 Division of Work

Partner jchen247 worked on most of the backbone, other words the motion of the enemy and the walls. Later on, we worked as a team to find bugs and fix them accordingly. Jchen247 worked on most of the logic, such as the enemy projectile and player projectile. Then again, we worked as a team to find out bugs and fixed them. And partner hlin42, worked on the timer interrupts, mothership, and other basic functionalities like end game, new level, RGB, and pause. We spent most of our time together in the lab, and utilized every lab hour. So, we did not spend most of our time writing our own code, we spent most of our time working together, and whenever one gets stuck, the other will jump in and take it from there.

1.8 Outside Materials

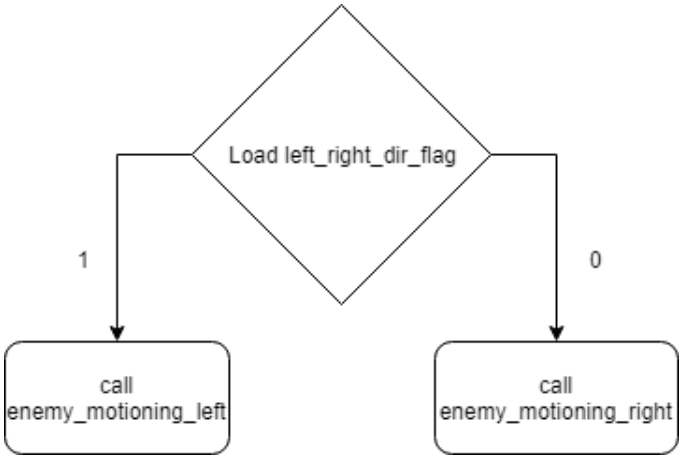
UM10120 Volume 1: LPC213x User Manual
www.asciitable.com
http://www.pacxon4u.com/space-invaders/

2 Final Flow Charts and Routines

Most of these routines are explained earlier with the memory address chart and description of each functionalities we have to implement. Here, the flowcharts explain for themselves. Our flowcharts are self explanatory, and goes in depth. Each bubble will explain what happens.

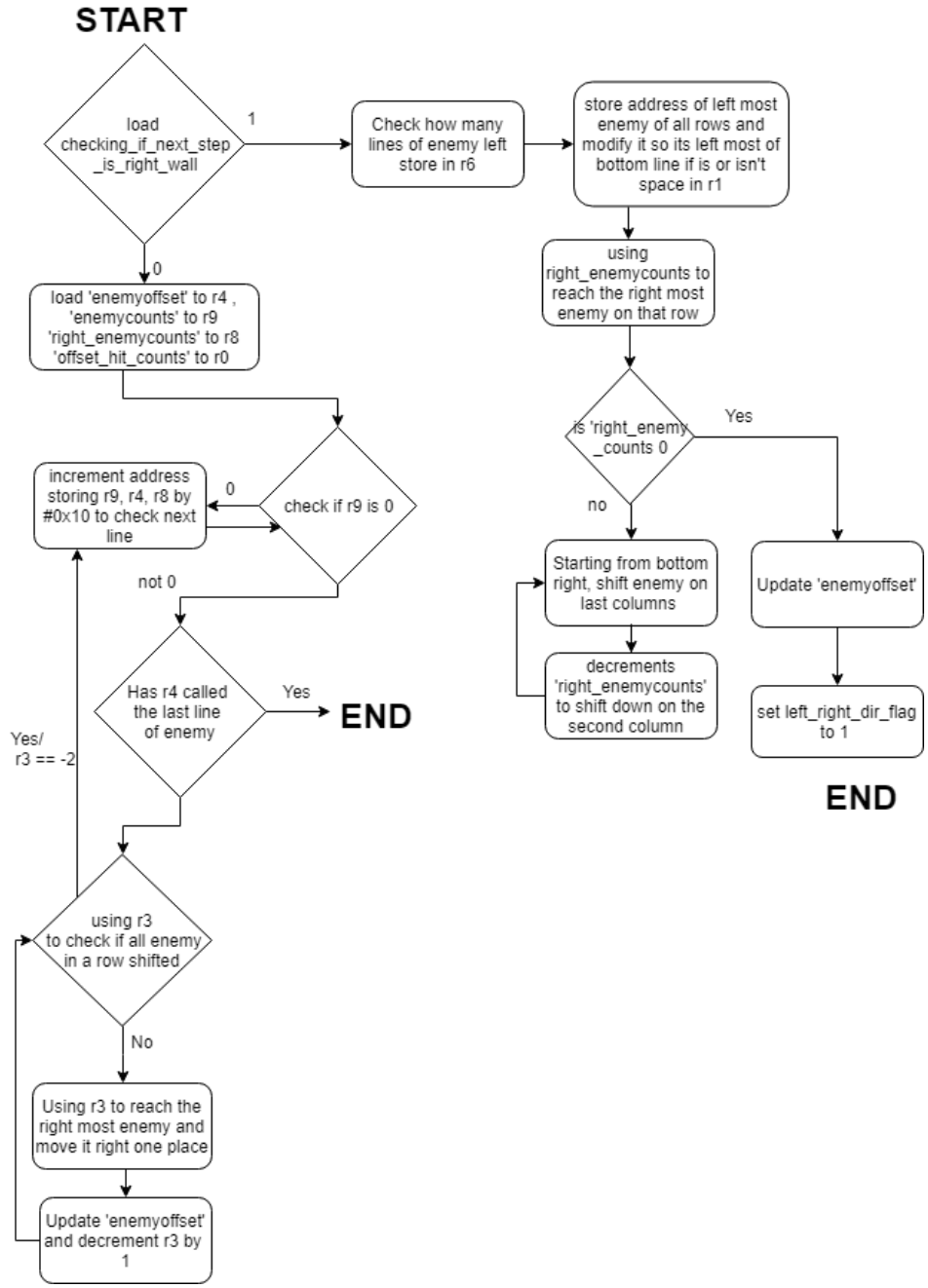
2.1 Enemy Motion

Inside enemy motion we have many subroutines. Here is our basic Enemy Motion routine.



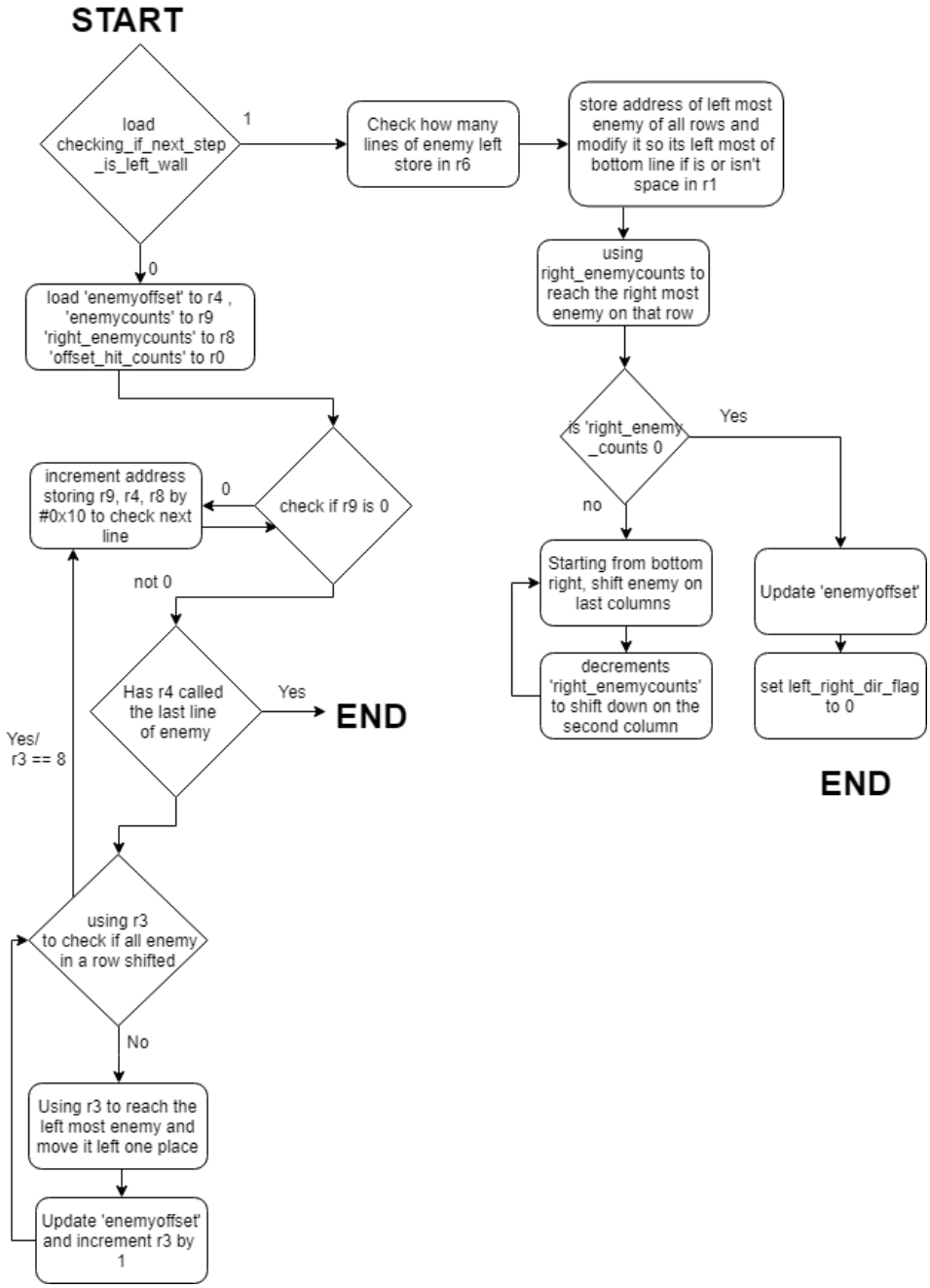
The general idea of this routine is we check to see if the left most enemy or right most enemy hits the wall. Then we write 1 or 0 into our memory address that value.

2.1.1 Enemy Right Motion



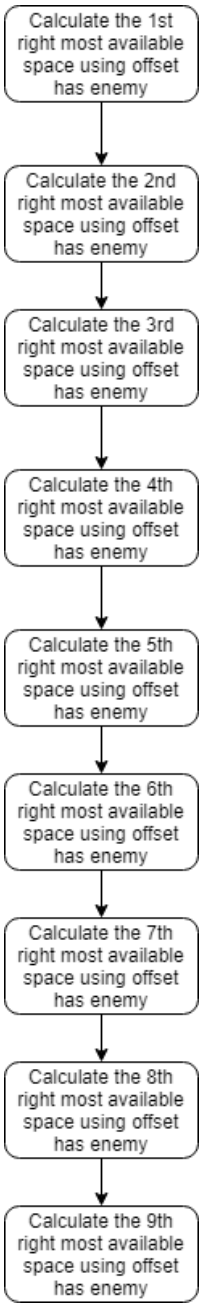
This flowchart is self-explanatory. We load the direction flag to see if the enemy has reached the right wall. If not, we change the offset, and rewrite the board so it looks like the enemy has shifted to the right. Then, if they reached the wall, every row of enemy invaders will shift down by 0x19.

2.1.2 Enemy Left Motion



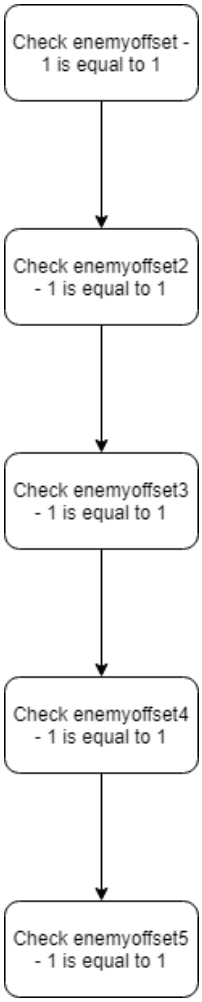
The logic for this is the exact same as Enemy Right Motion.

2.1.3 Reach Right Wall



We constantly check the outermost enemy, other words the enemy that is closest to the wall. This is used for our motion routine

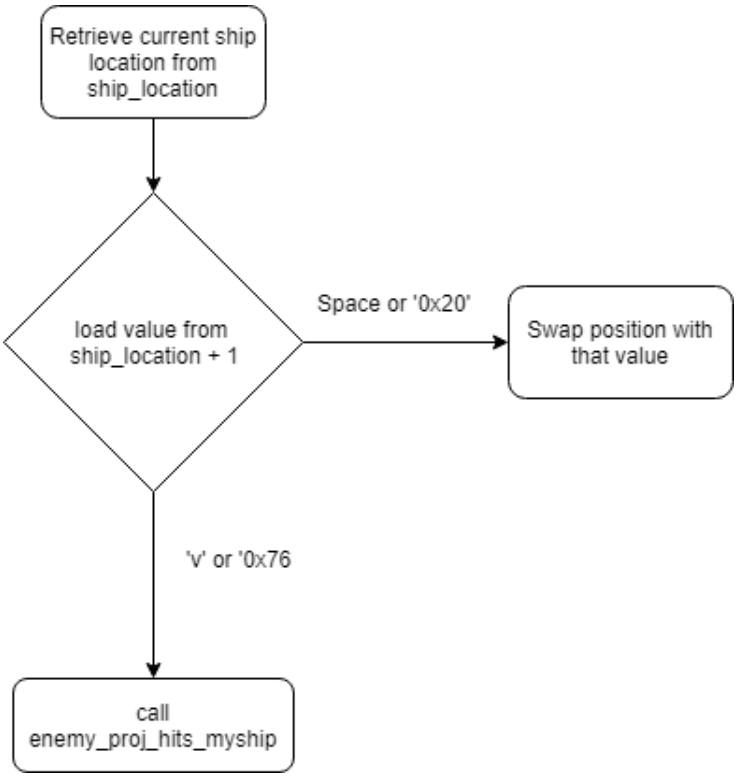
2.1.4 Reach Left Wall



Same logic behind reach right wall.

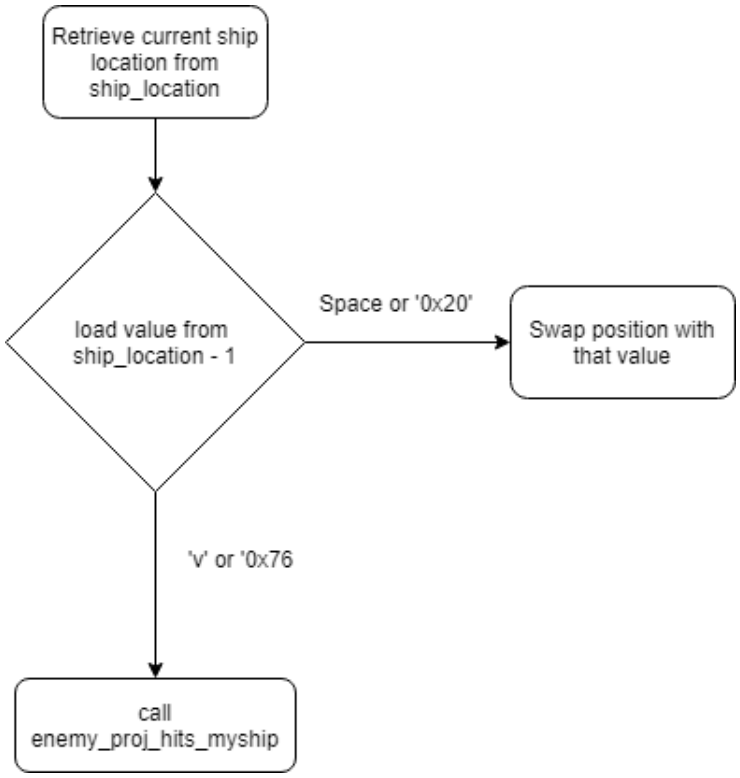
2.2 Player Ship Shifting

2.2.1 Player Shift Right



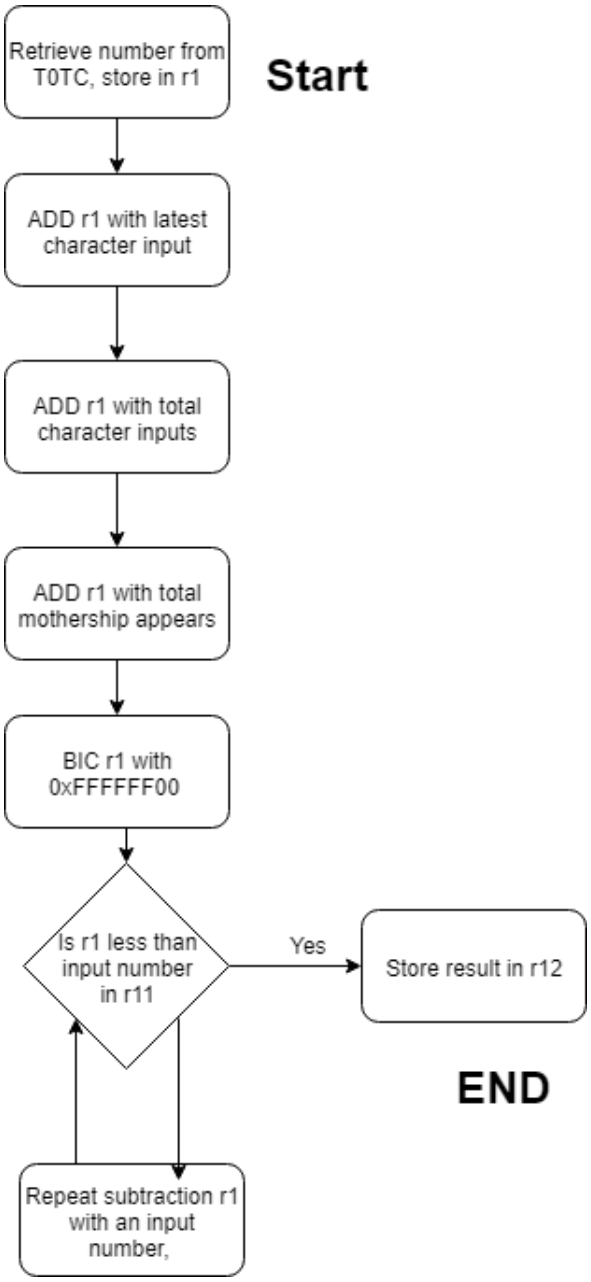
We find the ship current location. All the empty spaces on our board are spacebars, 0x20. Here we account for enemy projectile hitting our ship, which is handled in another routine. When shifting right, we swap position with the 0x20 right of ship location.

2.2.2 Player Shift Left



Same logic as player shift right.

2.3 Random Number Generator (RNG)

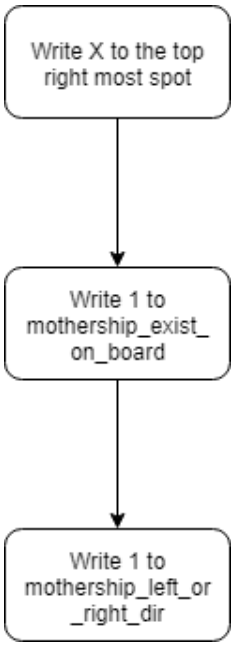


Our random number generator consists of many things. First the timer, the amount of mothership appeared, total of character inputs, and total number of specific character inputs. Since the machine itself cannot be random, what the user input is always going to be random, unless he or she presses the same key everytime. With those data, we do bit clears, and repeated subtraction to get the percentage we want.

2.4 Mothership

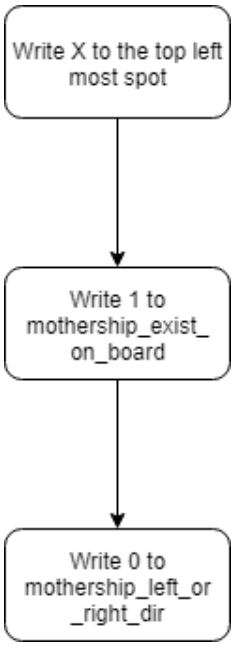
RNG will decide which side mothership will appear up from. The topmost row is reserved for the mothership, 'X'.

2.4.1 Mothership Shows Up From Right



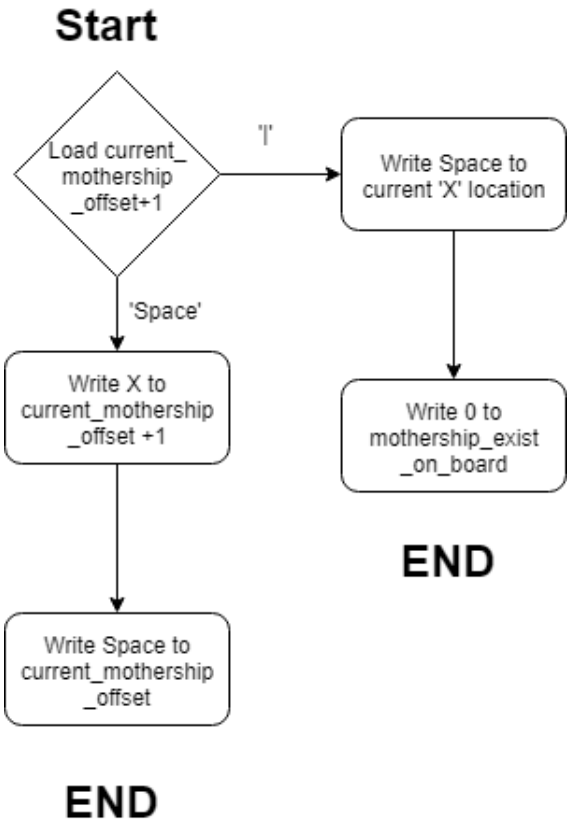
When mothership shows up, mothership on board will write an 1 into. The left and right direction flag will be set to 1 for right.

2.4.2 Mothership Shows Up From Left



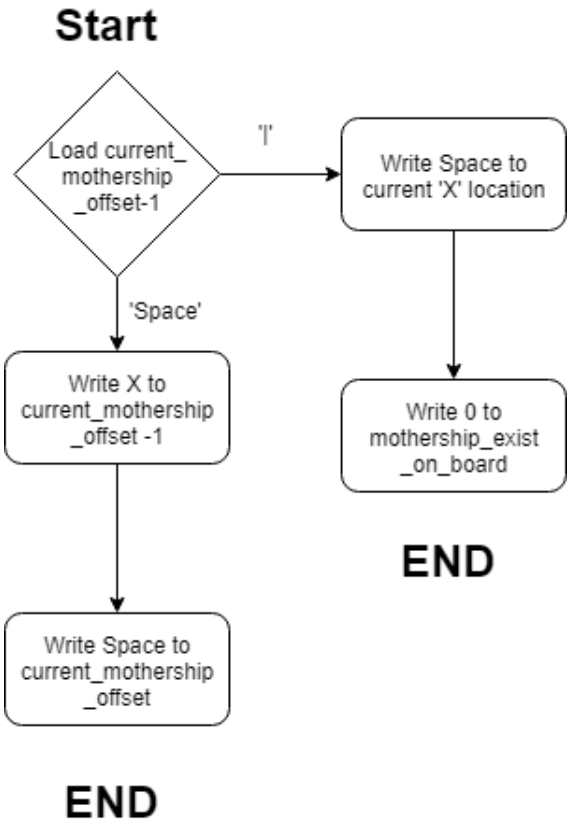
same logic as shows up from right. The difference here is instead of writing 1 in the direction flag we write 0 for left.

2.4.3 Mothership Right Motion



When motioning, we have to check for the wall, If it reaches the wall, it will disappear, and we write a 0 into the exist flag. If it is space, it will swap position with the space.

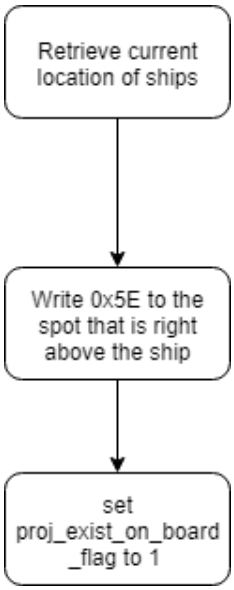
2.4.4 Mothership Left Motion



Same logic as right motion.

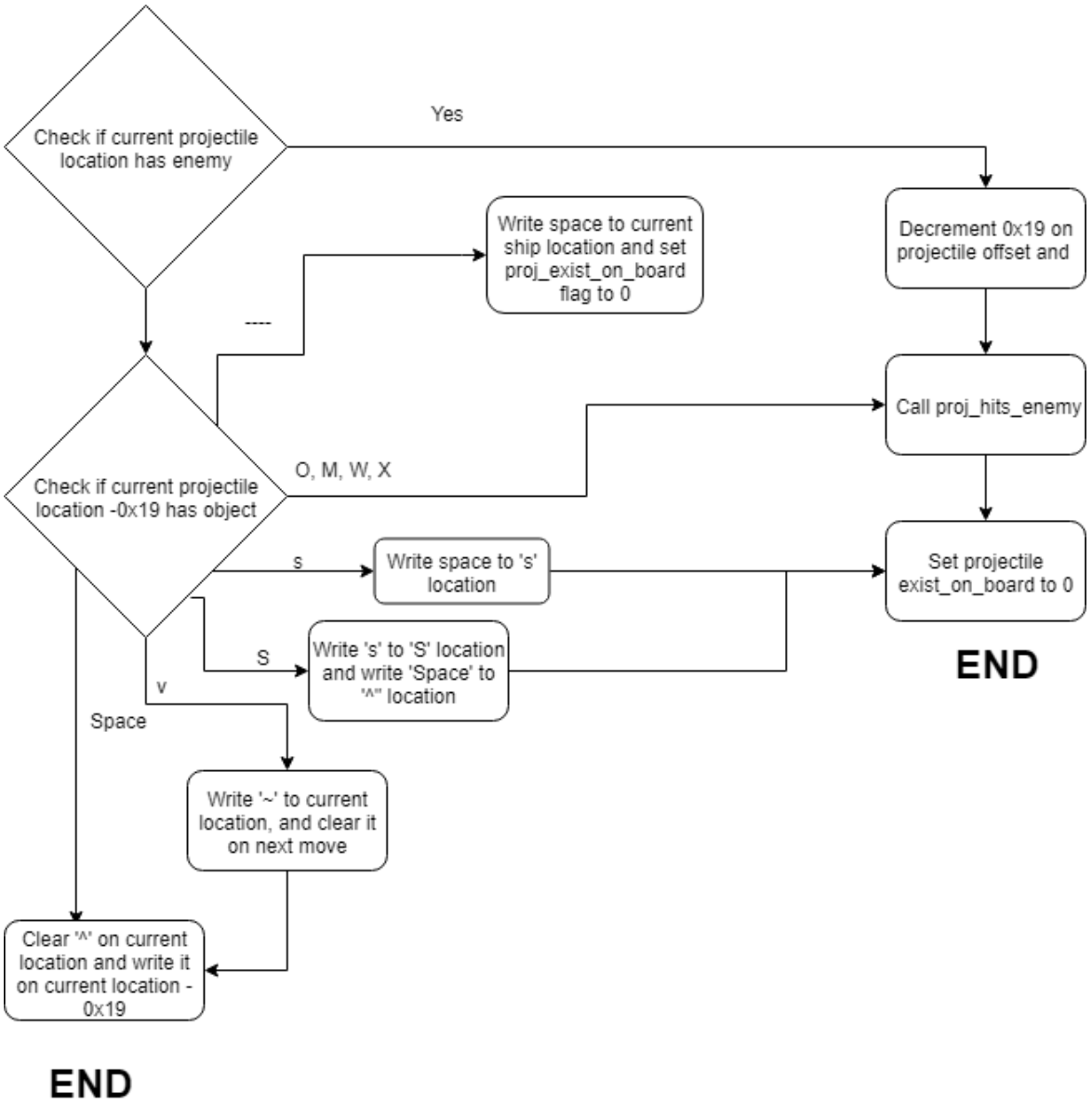
2.5 Player Projectiles

2.5.1 Player Projectile Eject



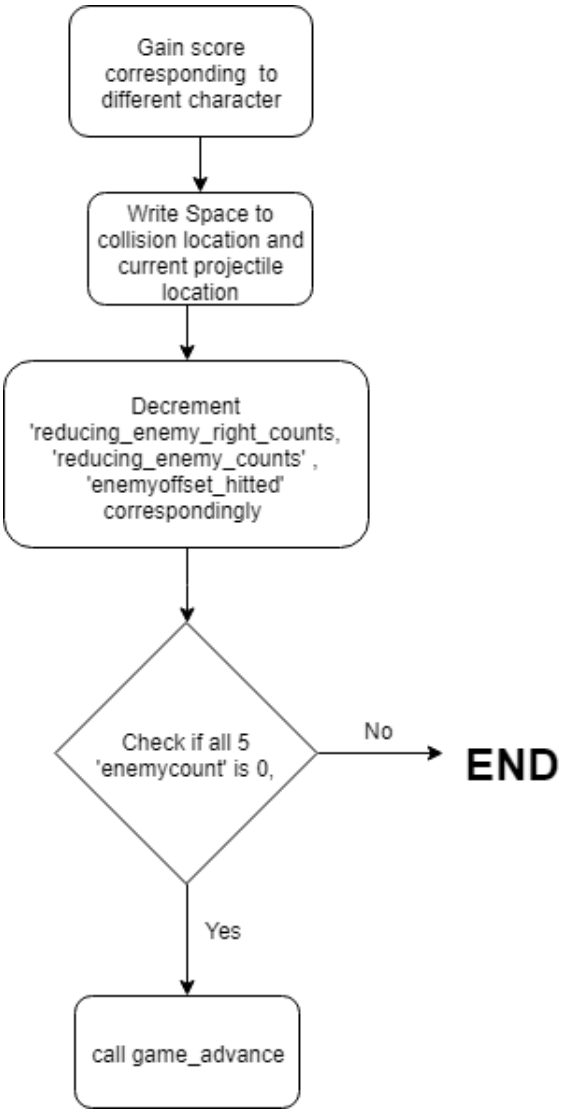
We check for where the current player ship is at. Then we write 0x5E, or '^' right above the ship, then set projectile exist to 1.

2.5.2 Player Projectile Motion



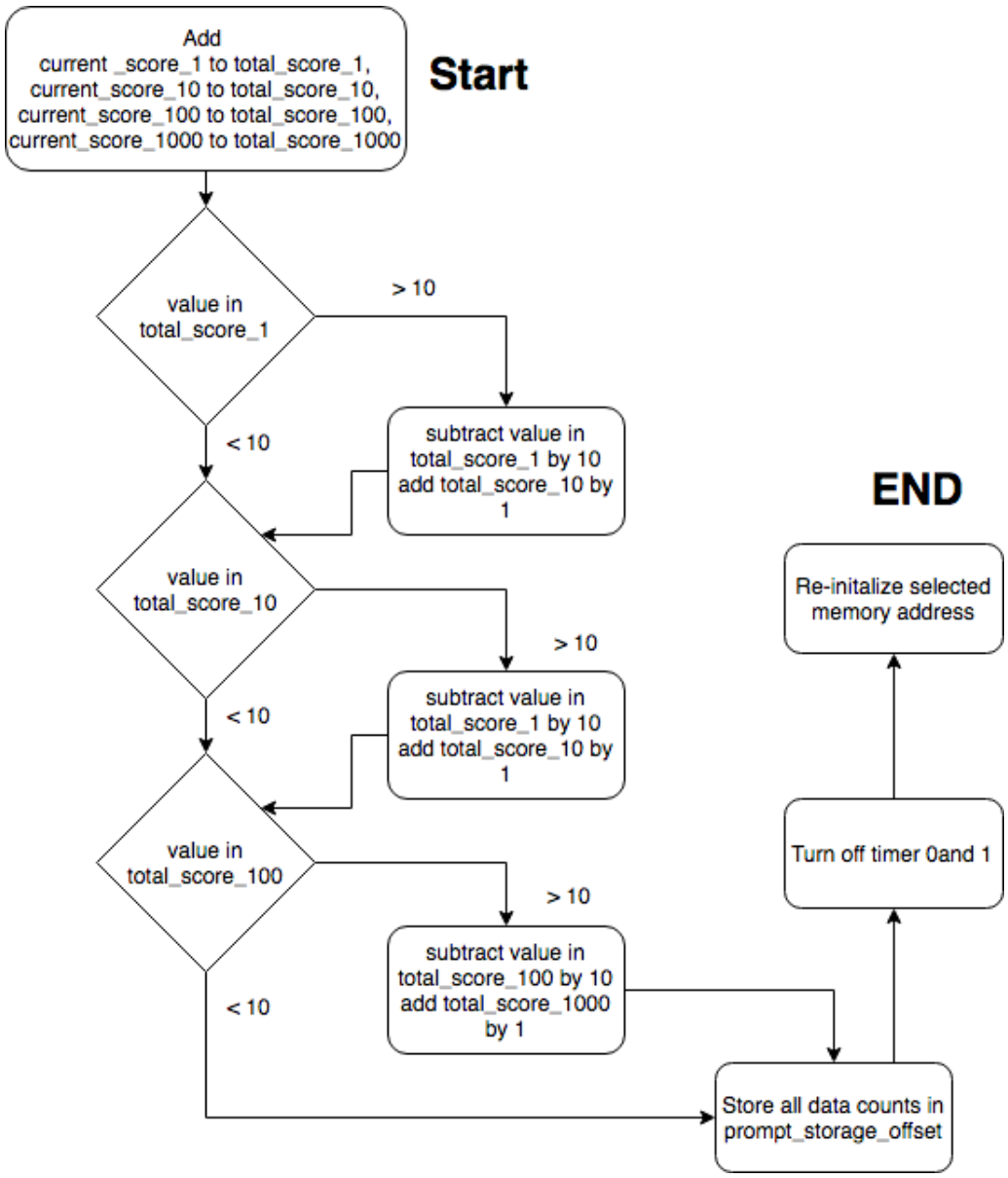
If the current location of our projectile contains an enemy. If there is, we will write a 0x20 at the projectile, call projectile hits an enemy, and write a 0 to its exists flag. Otherwise the projectile will keep moving forward until it hits something. If it hits an enemy invader, it will write a 0x20 at that location along with the projectile. If it hits the shield, 'S', it will become a weaker shield, 's'. Then weaker shield will become space. If it collides with enemy projectile, it will write a tilde, '~' then go through it until it hits something. When it hits, projectile exist will be 0.

2.5.3 Player Projectile Hits Enemy



When an enemy is hit, the player will gain corresponding points equal to the enemy hit, then the enemy will be overwritten with a space. Then we will reduce enemy right counts, enemy counts, and enemyoffset hit correspondingly. When the enemy count reaches zero, game will advance to next level.

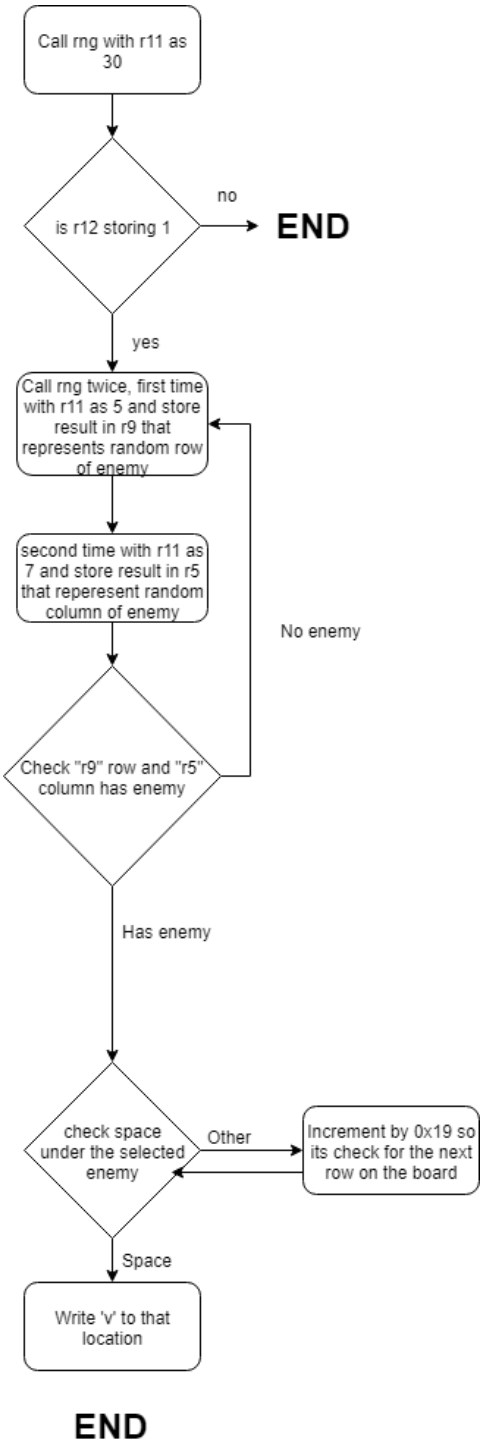
2.5.4 Game Advance / Next Level



When reaching new level, there are data that needs to be stored. The current score will be added on to the total score. Here, we did addition like common core. Using digits 0 to 9 instead of 0 to F. This will simplify conversion later on. And, during advancing, we will turn off both timers, and reinitialize the board and memory address for the new level. Note, total score will not be reset, current score will.

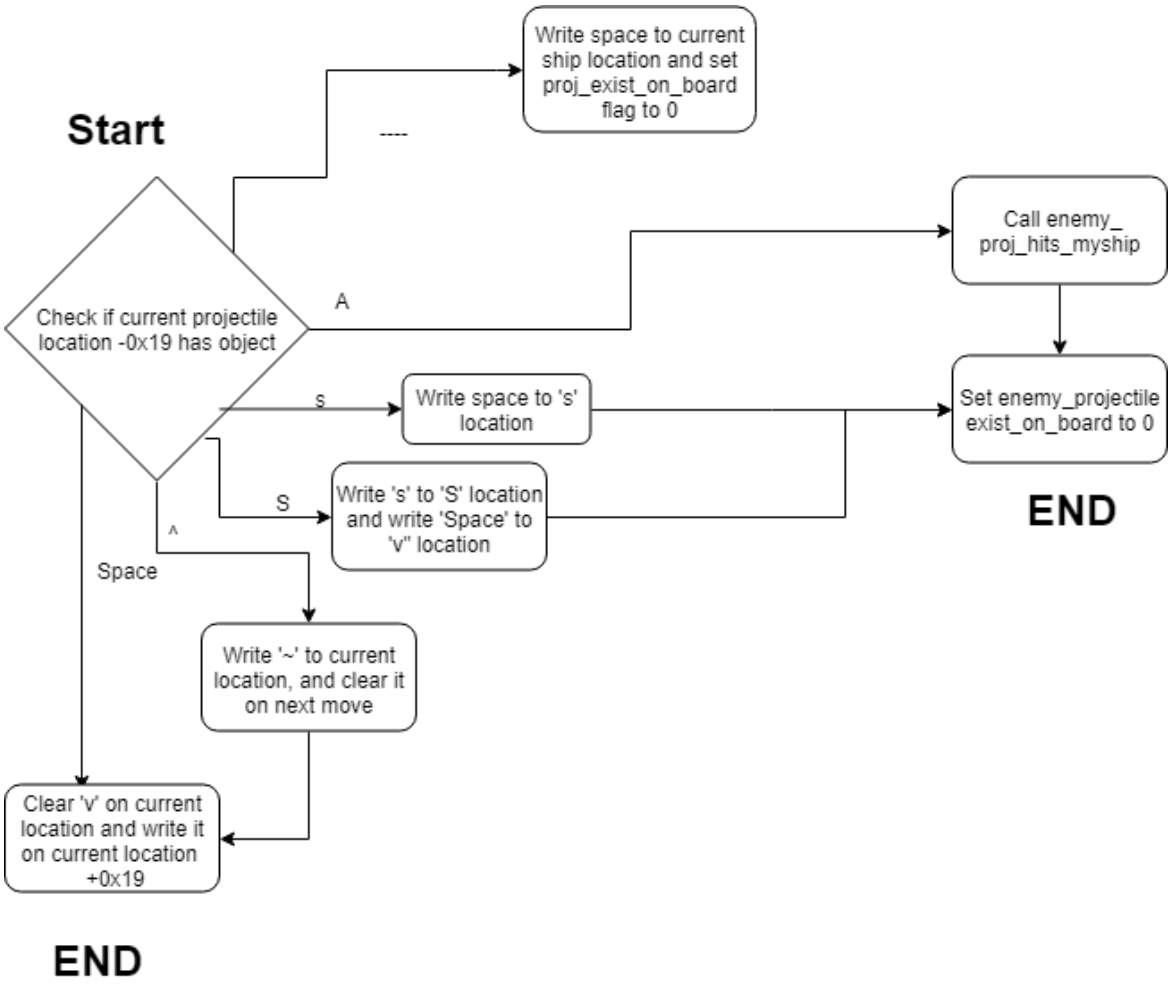
2.6 Enemy Invader Projectiles

2.6.1 Enemy Invader Projectile Eject



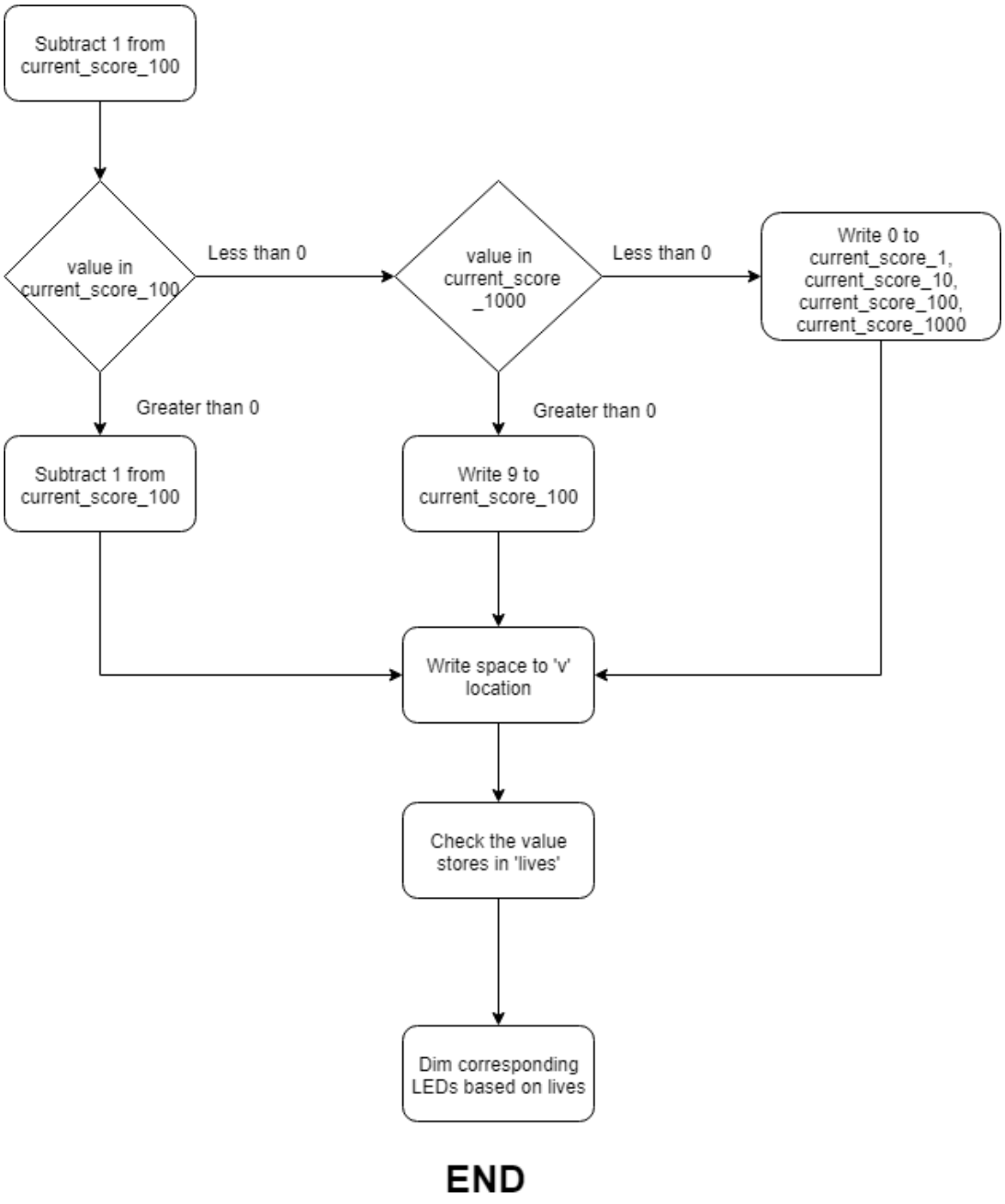
RNG will be used here to figure out when the enemy will shoot. As shown in the flowchart, the first two steps are the percentage rate. After that, RNG will be used again twice to figure out which row and column the enemy will shoot from. When these condition are met, a 'v' will appear, the exist flag will be 1, and advance until it hits something.

2.6.2 Enemy Invader Projectile Motion



This routine is similar to player projectile when in motion. When it hits a wall, a space will be written to the projectile location, and the exist flag will be 0. Hitting a shield will result it to be weaken or disappear. Hitting player projectile will result a tilde, and continue. Hitting player ship will result in a routine where enemy projectile hit player ship.

2.6.3 Invader Player Projectile Hits Player Ship

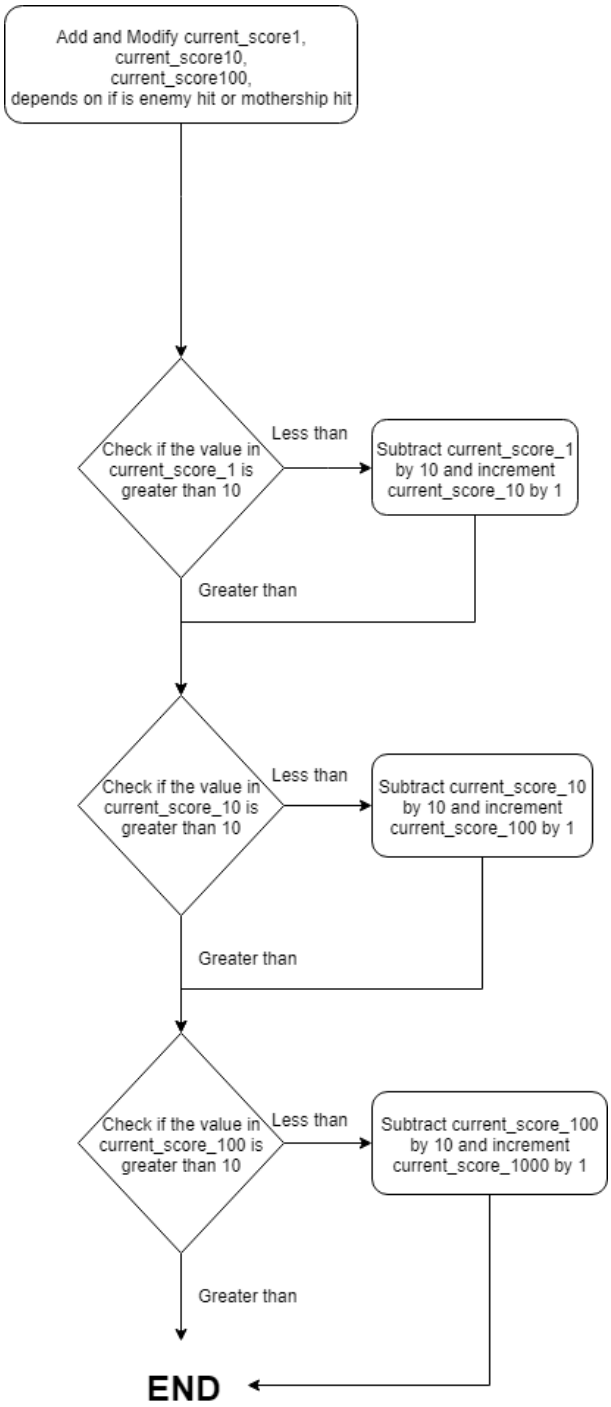


When enemy projectile hit player ship, one of the 4 LEDs will go off corresponding to the amount of lives the player has. And we subtract the score accordingly everytime the player was struck.

2.7 Scoring

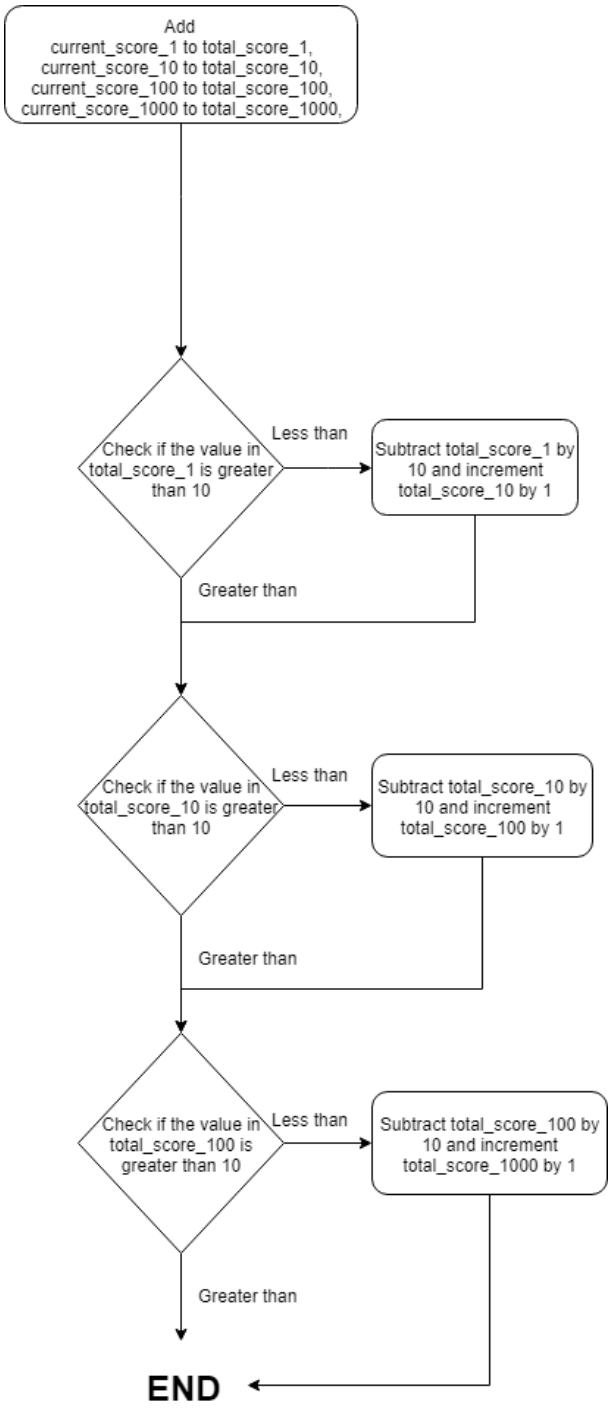
As explained before the current score will be added up like common core math. If the one's place is a 9, and we add a 1, it will become a 0, and we add one to the tens place. This is the basic idea for both adding to current score and total score.

2.7.1 Adding to Current Score



The current score will be displayed on the seven segment via strobing, which was done in the previous lab.

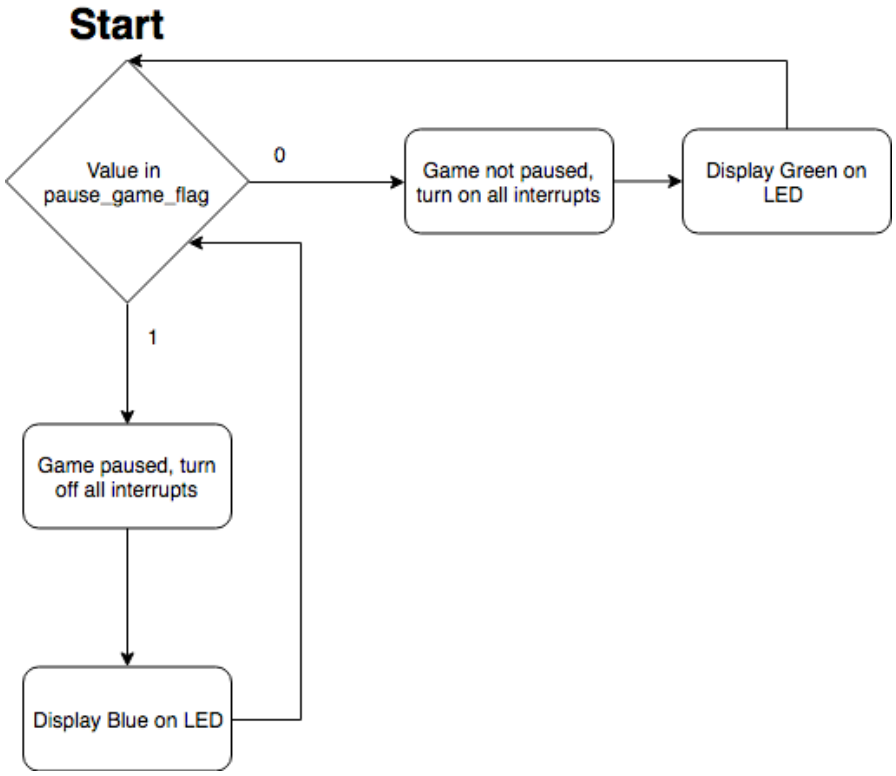
2.7.2 Adding to Total Score



2.8 Game Functions

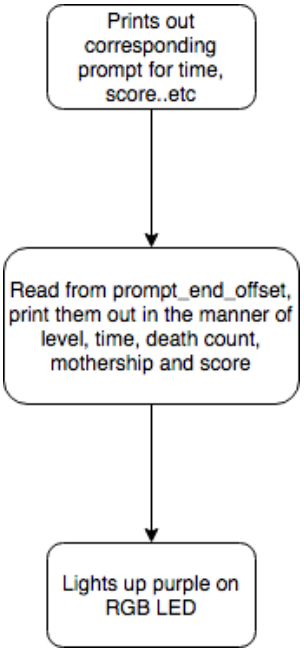
Some basic game functions are written in an outside loop.

2.8.1 Pausing and Resuming the Game



We check the pause status of the game when exiting the FIQ Handler. When the pause flag is 0, the timer and uart interrupt will be activated and game will continue its operation, and RGB LED will be green. If the flag is one, we turn off the timer interrupt and uart interrupt, and display blue on the RGB LED

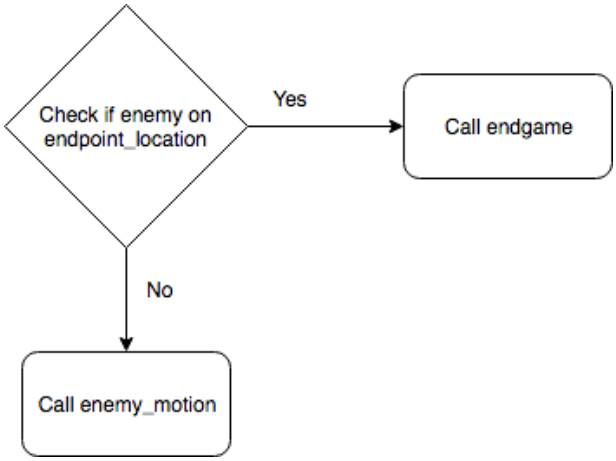
2.8.2 End Game



When ending a game, we will display a prompt that was manipulated. We read from the modified string and print it out on PuTTy, then the RGB LED will light purple.

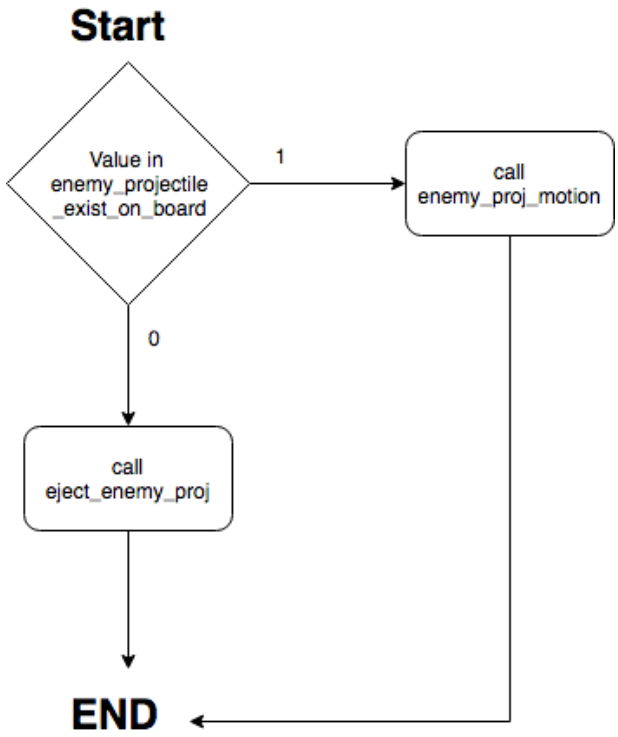
2.9 FIQ Interrupts

2.9.1 Timer 0



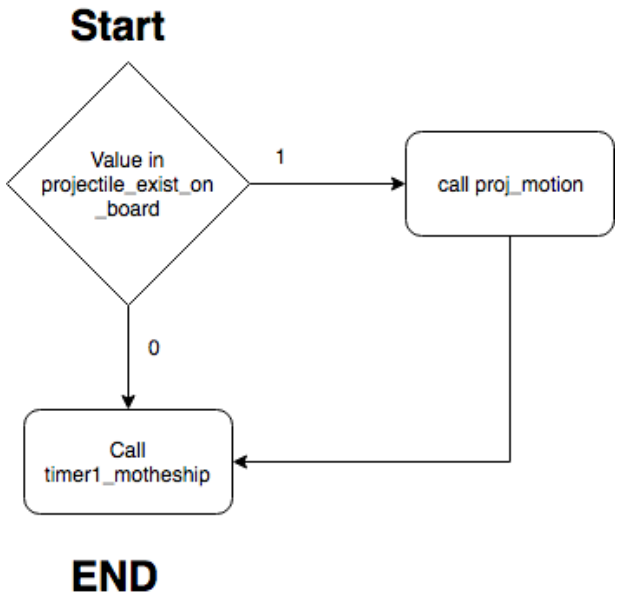
This timer was solely used for enemy motion. In our timer0 interrupt we check for enemy endpoint location if the enemy invaders has reached there. If yes, the game will end, otherwise enemy motion will commence.

2.9.2 Timer 1 (Enemy Projectile)



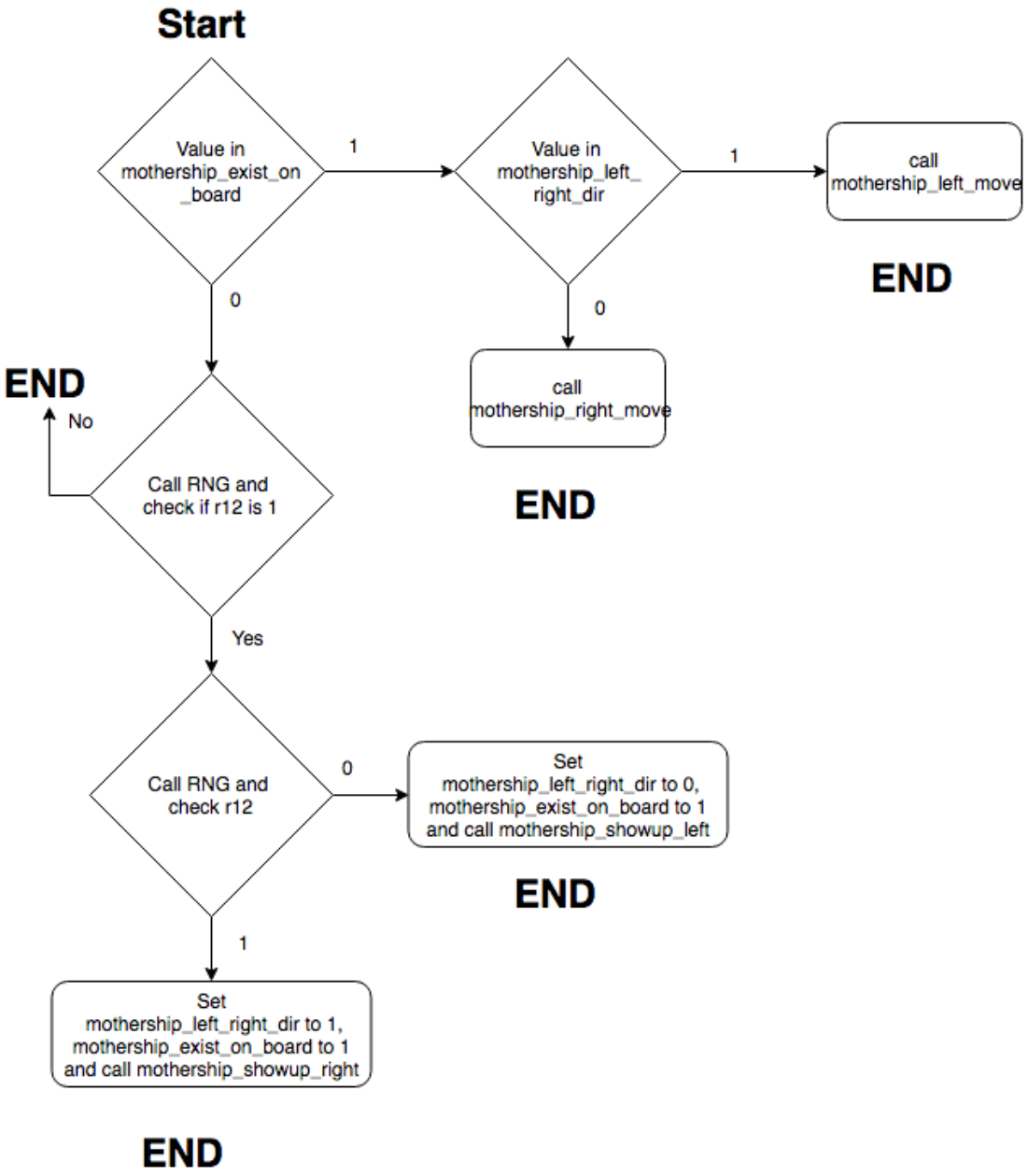
We check the enemy projectile exist flag. If 0, we eject enemy projectile based on RNG decision. If 1, it will call enemy projectile motion.

2.9.3 Timer 1 (Player Ship Projectile)



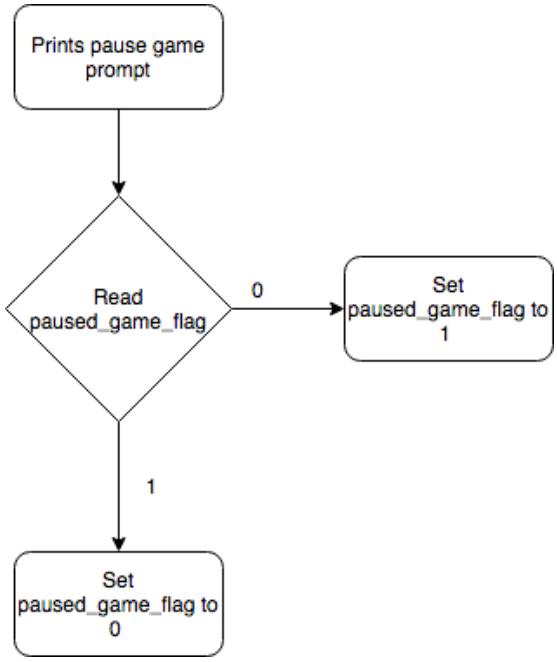
This is the same logic as enemy projectile in the timer. If exist is 1, we call projectile motion. If 0, the next routine will be called, which is the mothership. If the mothership exist, it will call the mothership motion.

2.9.4 Timer 1 (Mothership Motion)



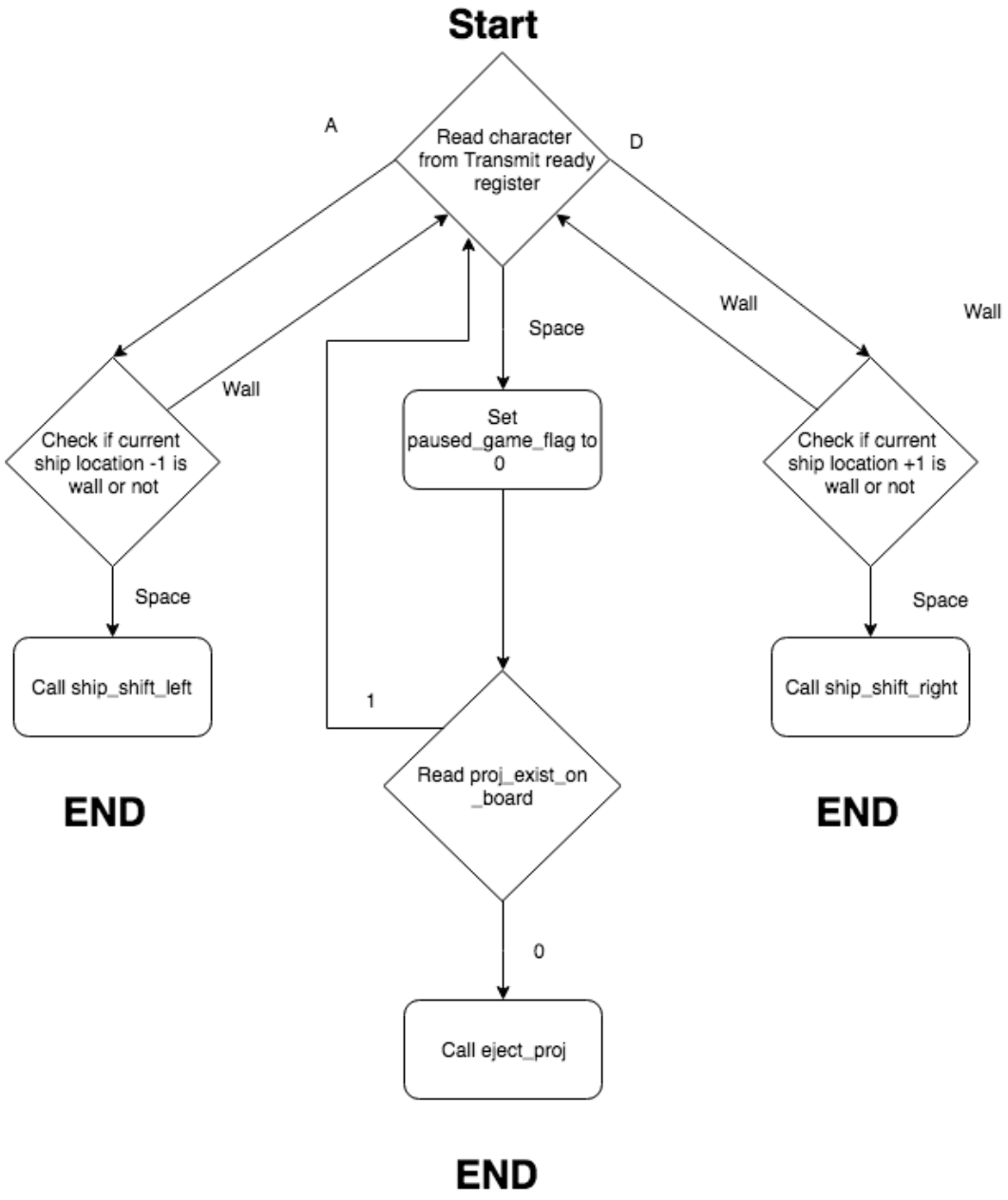
Here RNG will decide if the mothership will appear and if the mothership will come up from the left or right. If exist flag is 1, direction flag will decide left or right. Then, the timer will begin for mothership movement. Here the RNG will decide mothership existence. Timer will be solely used for motion.

2.9.5 External Interrupt (EINT1)



Pressing the P0.14 button will alternate between 0 and 1. Game will start paused, which equates to a paused game flag of 1. Pressing the button will turn it to 0, which will resume the game. If its 0, pressing it will result a 1, hence pausing the game.

2.9.6 UART Interrupt (UART0)



Here we read the transmit ready register data, and decide what it will do. If 'A' or 'D' or spacebar is pressed, it will run its corresponding routine, which is left movement, right movement, or eject projectile, respectively. If moving to a wall, it will check, and stop it.

2.10 Other routines

Other routines including read string, transmit character and other routines, may or may not have been used. Refer to previous documentation for these routines.

3 Conclusion

This project was our hardest project by far. We started the first day it was announced, and expected to finish within the 20 points extra credit. But many mishaps has occurred. Especially exception handler. We had too many undefine exception and data abort exception. Sometimes it would completely ignore lines of code in Keil. We used many ways to fix these problems, but later on, it happened again. We figured out using less BL will help. And indeed it fixed our problems. We did not need to use anymore LTORGs, since it was giving us warnings.

It was a great learning experience for us. Facing unexpected challenges, and resolving them. We learned a lot from our mistakes. Hence, that is why it took us the FULL time to do this project. In the end, we finished in time with our program working pretty flawlessly.

END