Wee Dig Dug

by

Christian Evans

Wee Dig Dug

Objective:

Design an assembly language program that recreates the classic Atari game Dig Dug using all of the concepts learned throughout the course and previous labs.

Instructions of Use:

To utilize the program, first be sure that the LPC2138 processor is configured properly according to the user manual, setting up the Universal Asynchronous Receiver/Transmitter (UART0) and General Purpose Input/Output (GPIO). Execute PuTTY and load the serial port at which the UART is connected (ports in use can be assessed using the [mode] command in cmd). Set the speed to 115,200. Once opened, run the program.

You will be prompted with a couple pages of instructions. Press the ENTER key once each has been read. The game will begin once the board is displayed. The game clock will be set to 120 seconds. You can move up (w), down (s), left (a), or right (d) using the parenthesized keys. You may also use the spacebar to attack enemies that you have an open path to. Coming in contact with an enemy will cause you to lose a life. Once all of your lives have been depleted or the time runs out, the game will end. You may play again by pressing ENTER or quit by pressing ‘q’. All other key input throughout the game will be ignored.

You will notice key game information being displayed on the GPIO board as well. The seven-segment display will show you the current level you are on, the LEDs will display the number of lives you have left, and the RGB LED will display various game states: green (game running), blue (game paused), purple (game over), and red (attack in progress). You may also use the button labeled P0.14 to pause the game at any time.

Debugging:

The debugging steps began with setting breakpoints at subroutine labels and stepping through the program from there. Monitoring the local register values, I attained invaluable information regarding the program and minor runtime errors encountered. I was able to diagnose the issues by taking into account desired program direction versus actual paths taken. Upon seeing the program jump to unexpected points and store careless values in registers, appropriate action was taken to correct the issues and note coding mishaps for further understanding. Using the initial flowchart and checklist, I maintained a set path on what was next to be accomplished and remained focused on that area until it was finished. I would routinely write small portions of code and test it before continuing as to not bury my progress behind a lost mistake made long ago.

References:

- ARM Architecture Reference Manual, 2nd Edition, Addison-Wesley, 1996-2000
- Kris Schindler, Introduction to Microprocessor Based Sytems Using the ARM Microprocessor, 2nd Edition, Pearson, 2013
- ARM PrimeCellTM Real Time Clock (PL031) Technical Reference Manual, ARM Limited, 2001
- ARM PrimeCellTM General Purpose Input/Output (PL061) Technical Reference Manual, ARM Limited, 2000

- ARM UART Notes, Dr. Dick Blandford, EE 354, University of Evansville, Fall 2011
- UM10120 Volume 1: LPC213x User Manual, Koninklijke Philips Electronics N.V., 24 June 2005

How It Works:

The code is run from the basis of a c file called lab7wrapper.c. It calls the initialization subroutine to initialize the UART, GPIO, and PINSEL0/1. The lab7 subroutine is then called. Extensively utilizing the output_string subroutine throughout the program, prompts are displayed via PuTTY, as well as user input.

Once the game is triggered to begin, interrupts are enabled with a timer set to interrupt every tenth of a second. Pressing the w, a, s, or d keys will set a flag stored in memory to later tell the program to update the user coordinates in either the up, left, down, or right directions respectively. Hitting the spacebar will likewise set a flag indicating an attack to take place. The push button (P0.14) is initialized to interrupt the program every time it is pressed. Handling this interrupt will check a flag set in memory to see if the game is paused and update the flag accordingly whether to unpause or pause the game. These flags will be checked throughout different portions of the timer interrupt.

When the timer interrupts the program, it will check whether or not the game is paused, displaying a flashing screen indicating the state if so. Elsewise, the interrupt continues, first checking the game clock and updating it every ten tics (to complete a full second of gameplay). Once the game clock has been updated, the program checks for collision between the user and each of the enemies, decreasing the number of lives and resetting the user location if applicable. The user movement flag is then checked, determining what character to display (based upon direction) and whether or not to update the coordinates. Next, the attack flag is checked and will determine whether or not an enemy was defeated with the attack and will display the character when possible.

Score will update accordingly through movement and attacks in these first two phases. After the enemies are all possibly defeated, the game checks the lively status of each enemy to determine whether or not the game should level up (setting a flag if so). Likewise, if all lives are depleted then a flag will be set to indicate so once the interrupt is handled.

Each enemy location/direction is updated based on possible movement and random factors. Upon exiting the handler, the main subroutine (lab7) will continue to run until the next interrupt occurs. This subroutine will check the level up and game over flags to determine if any sort of restart should occur and how exactly it should be handled.

More detailed information exists within the short descriptions and flowcharts below.

lab7:

Initializes all game factors and continuously loops to allow for further interrupts until certain conditions/flags are set.

init_param:

Utilizing stored random numbers, sets initial enemy locations and characteristics.

init_display:

Modifies the preset board string to account for the information acquired from init_param, indicating enemy spawn locations.

display_board:

Loads all coordinate information for the user and enemies, rewriting the board based on the values stored in memory.

board_reset:

Reverts the board to its initial prewritten state.

collision_adjust:

Compares argument coordinates to determine if a collision occurred. Updates the number of lives remaining.

reinitialize:

Resets all flags and coordinates back to initial values for new game or level.

end_initialize:

Reverts score, time left, lives, level, and refresh rate back to initial values.

FIQ_Handler:

Handles all interrupts from the UART0, ENT1 (push button), and Timer0, clearing each if they set.

random_number:

Loads current value from Timer1 and stores it into memory

char_check:

Checks if the character at a location on the board is a variation of a user character

enemy_check:

Checks if the character at a given location is that of an enemy, also comparing the coordinates of each of the enemies to determine if the character is to be overwritten or not.

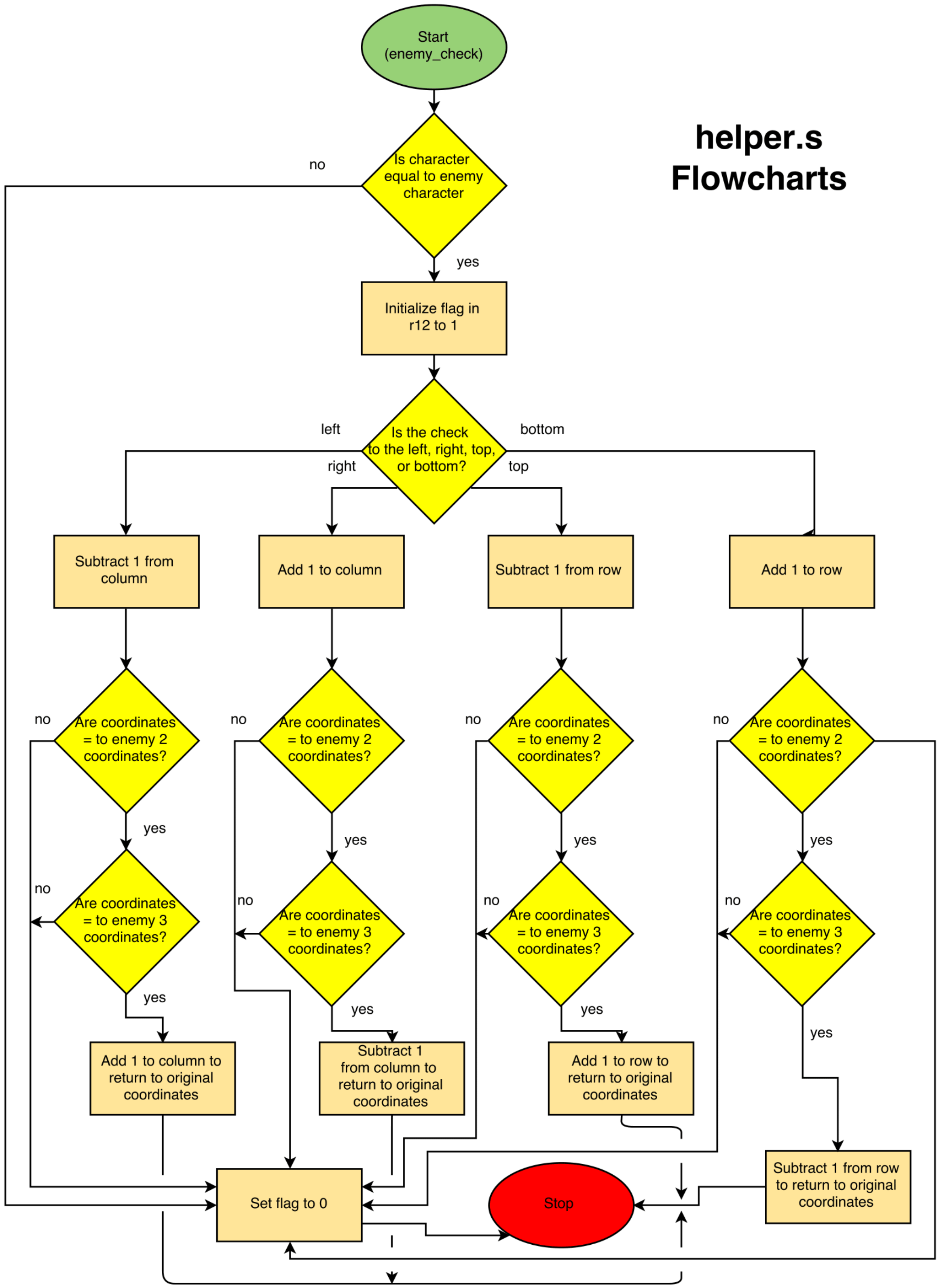
spawn_check:

Compares the random spawning locations of the enemies to deter spawn collision, changing the coordinates if necessary.

move_check:

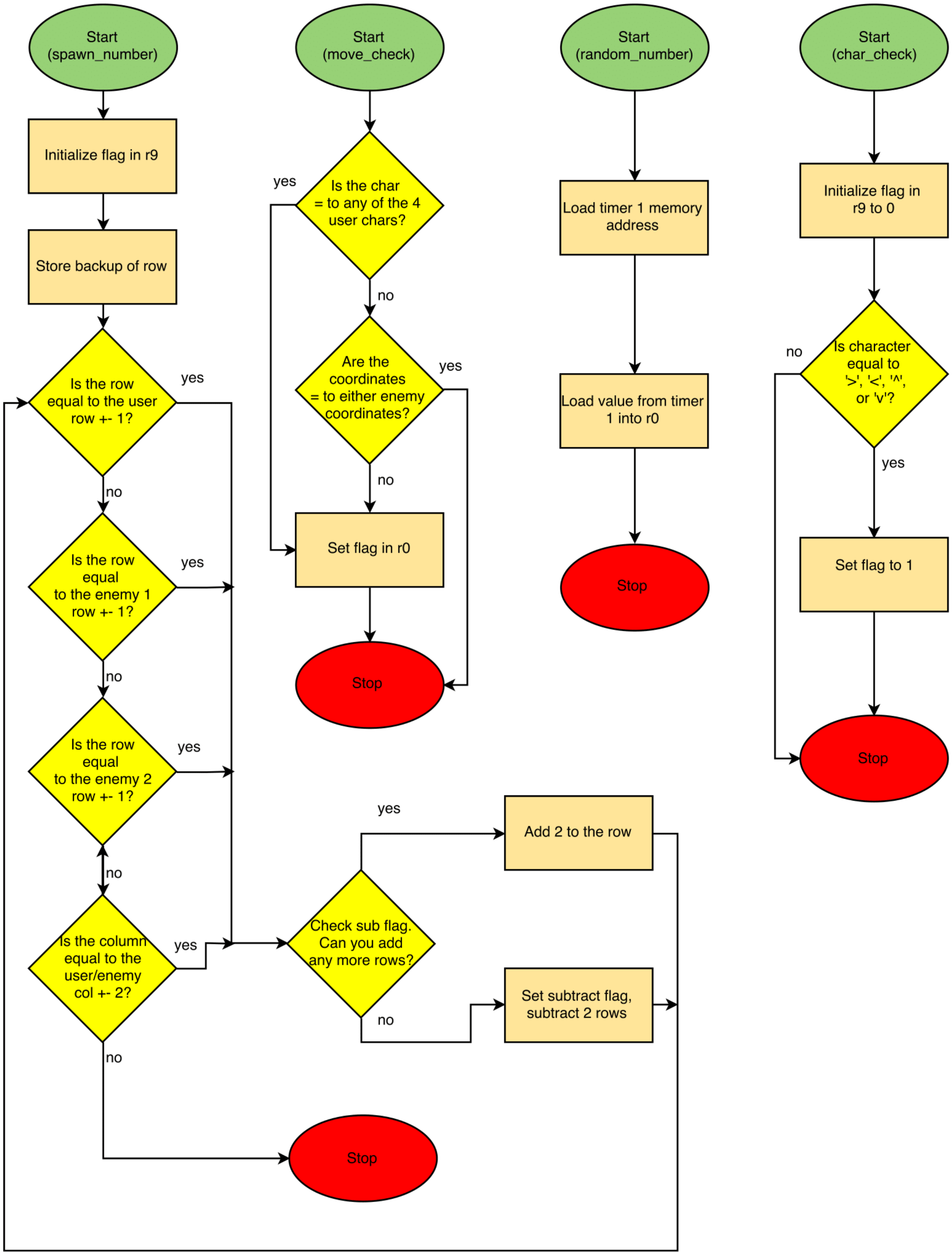
Checks the parameterized board location to determine if the enemy movement is plausible.

- Attached are the flowcharts demonstrating the written steps

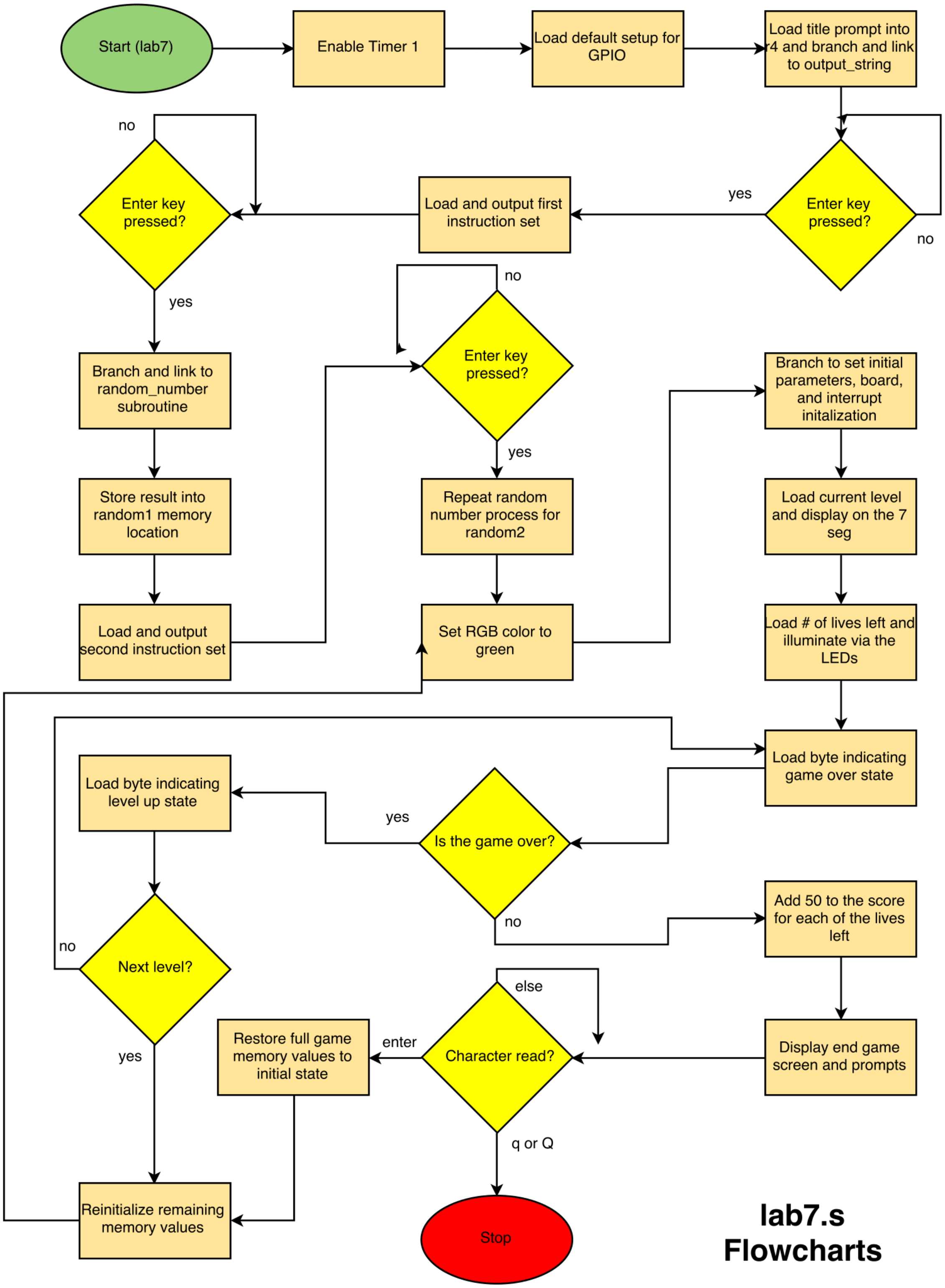

Start
(enemy_check)
Is character equal to enemy character
no
yes
Initialize flag in r12 to 1
Is the check to the left, right, top, or bottom?
left
right
top
bottom
Subtract 1 from column
Add 1 to column
Subtract 1 from row
Add 1 to row
Are coordinates = to enemy 2 coordinates?
Are coordinates = to enemy 3 coordinates?
no
yes
Add 1 to column to return to original coordinates
Subtract 1 from column to return to original coordinates
Add 1 to row to return to original coordinates
Subtract 1 from row to return to original coordinates
Set flag to 0
Stop

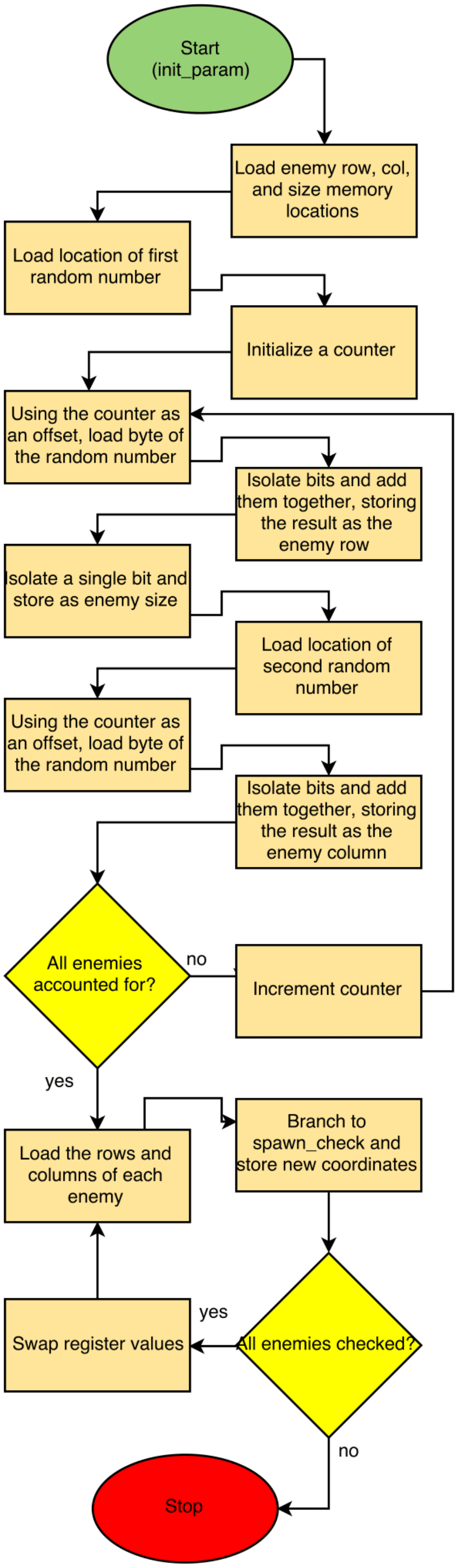
helper.s Flowcharts

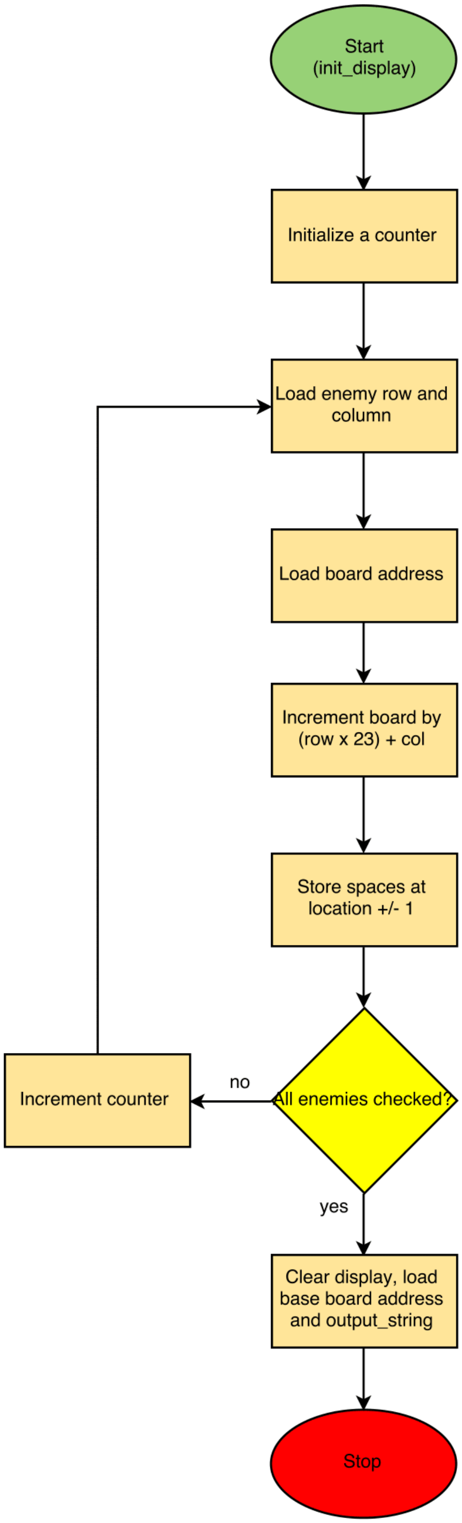
Wee Dig Dug


Start (spawn_number)
Initialize flag in r9
Store backup of row
Is the row equal to the user row +- 1?
yes
no
Is the row equal to the enemy 1 row +- 1?
yes
no
Is the row equal to the enemy 2 row +- 1?
yes
no
Is the column equal to the user/enemy col +- 2?
yes
no
Check sub flag. Can you add any more rows?
yes
Add 2 to the row
no
Set subtract flag, subtract 2 rows
Stop
Start (move_check)
Is the char = to any of the 4 user chars?
yes
no
Are the coordinates = to either enemy coordinates?
yes
no
Set flag in r0
Stop
Start (random_number)
Load timer 1 memory address
Load value from timer 1 into r0
Stop
Start (char_check)
Initialize flag in r9 to 0
Is character equal to '>', '<', '^', or 'v'?
no
yes
Set flag to 1
Stop

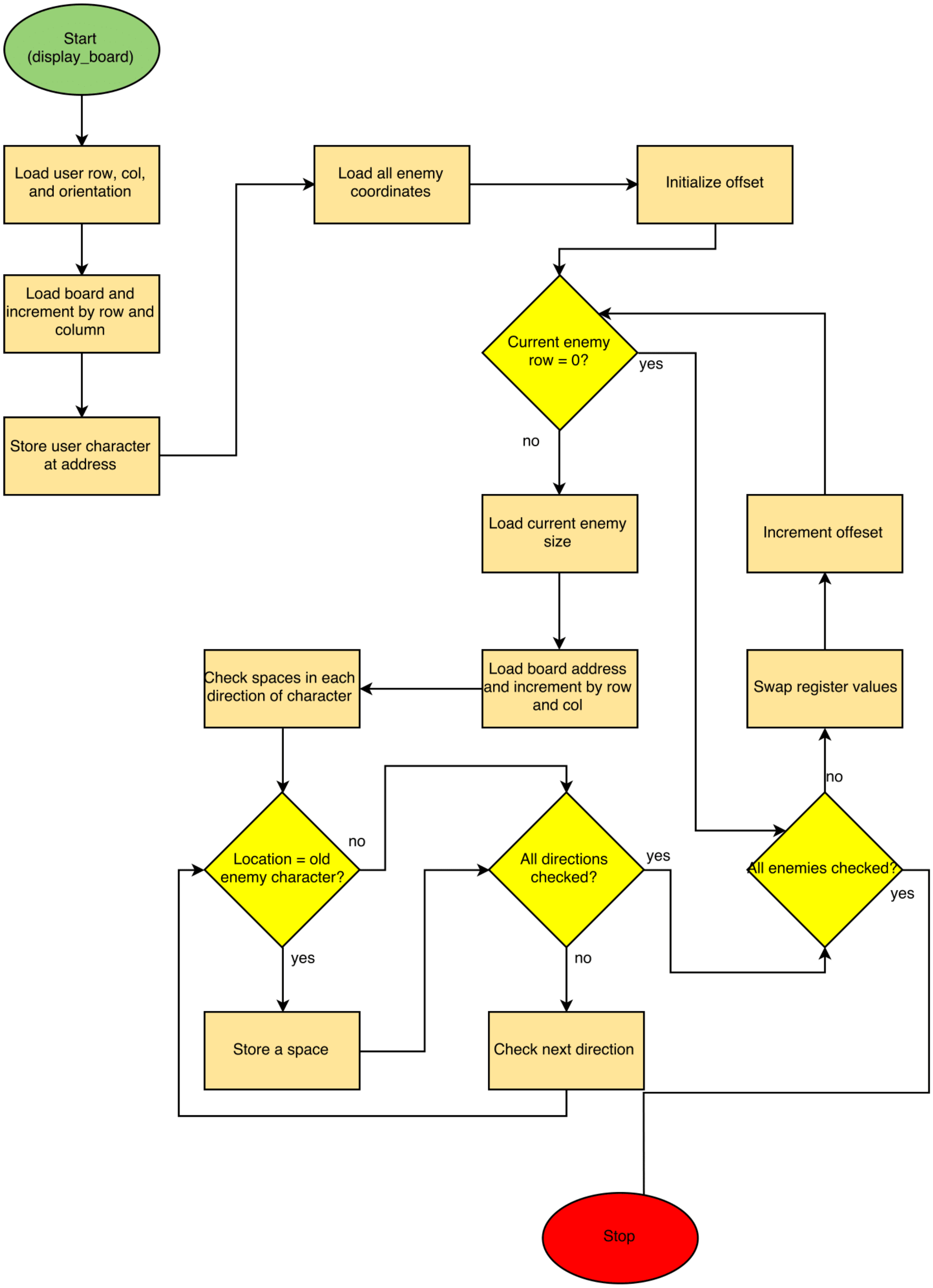
Wee Dig Dug


Start (lab7)
Enable Timer 1
Load default setup for GPIO
Load title prompt into r4 and branch and link to output_string
Enter key pressed?
no
yes
Load and output first instruction set
Enter key pressed?
no
yes
Branch and link to random_number subroutine
Store result into random1 memory location
Load and output second instruction set
Enter key pressed?
no
yes
Repeat random number process for random2
Set RGB color to green
Branch to set initial parameters, board, and interrupt initalization
Load current level and display on the 7 seg
Load # of lives left and illuminate via the LEDs
Load byte indicating game over state
Is the game over?
yes
no
Load byte indicating level up state
Next level?
no
yes
Add 50 to the score for each of the lives left
Display end game screen and prompts
Character read?
else
enter
q or Q
Restore full game memory values to initial state
Reinitialize remaining memory values
Stop
lab7.s Flowcharts

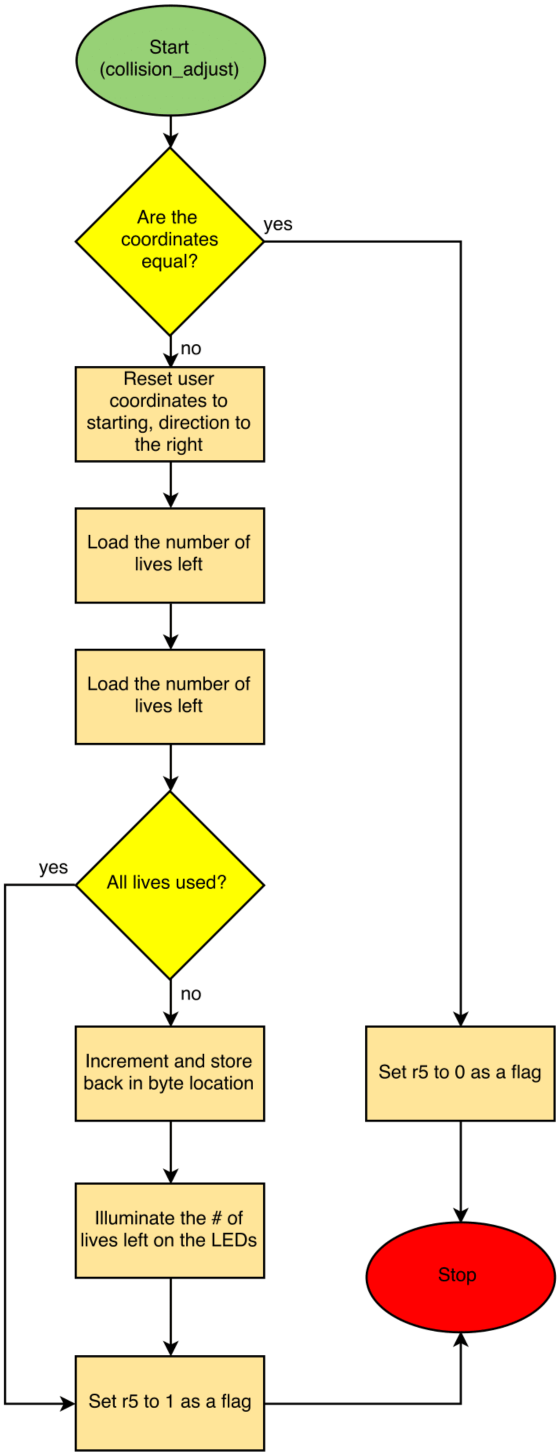

Start
(init_param)
Load enemy row, col, and size memory locations
Load location of first random number
Initialize a counter
Using the counter as an offset, load byte of the random number
Isolate bits and add them together, storing the result as the enemy row
Isolate a single bit and store as enemy size
Load location of second random number
Using the counter as an offset, load byte of the random number
Isolate bits and add them together, storing the result as the enemy column
All enemies accounted for?
no
Increment counter
yes
Load the rows and columns of each enemy
Branch to spawn_check and store new coordinates
All enemies checked?
yes
Swap register values
no
Stop

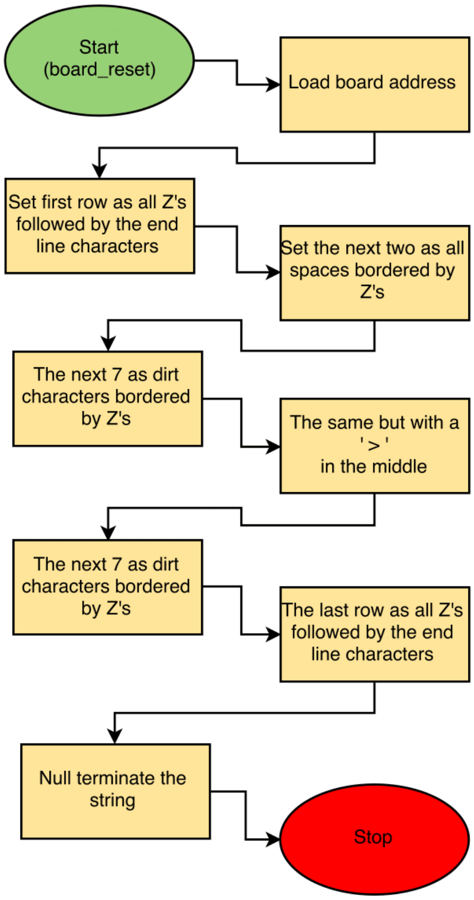

Start
(init_display)
Initialize a counter
Load enemy row and column
Load board address
Increment board by (row x 23) + col
Store spaces at location +/- 1
All enemies checked?
no
Increment counter
yes
Clear display, load base board address and output_string
Stop

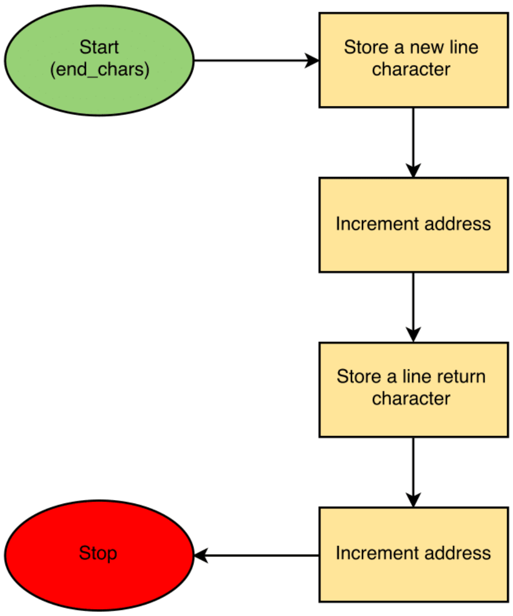
Wee Dig Dug


Start (display_board)
Load user row, col, and orientation
Load board and increment by row and column
Store user character at address
Load all enemy coordinates
Initialize offset
Current enemy row = 0?
yes
no
Load current enemy size
Load board address and increment by row and col
Check spaces in each direction of character
Location = old enemy character?
no
yes
Store a space
All directions checked?
yes
no
Check next direction
Increment offeset
Swap register values
no
All enemies checked?
yes
Stop

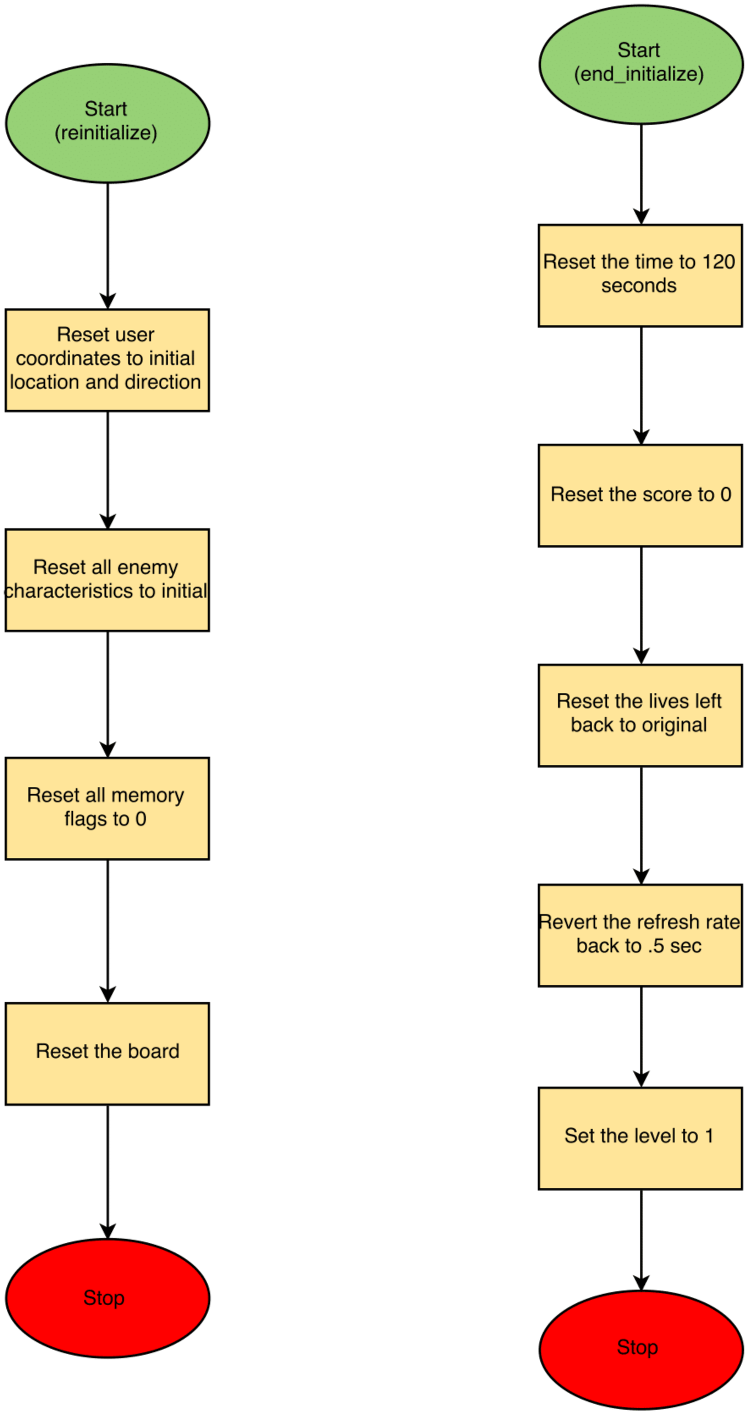
Wee Dig Dug

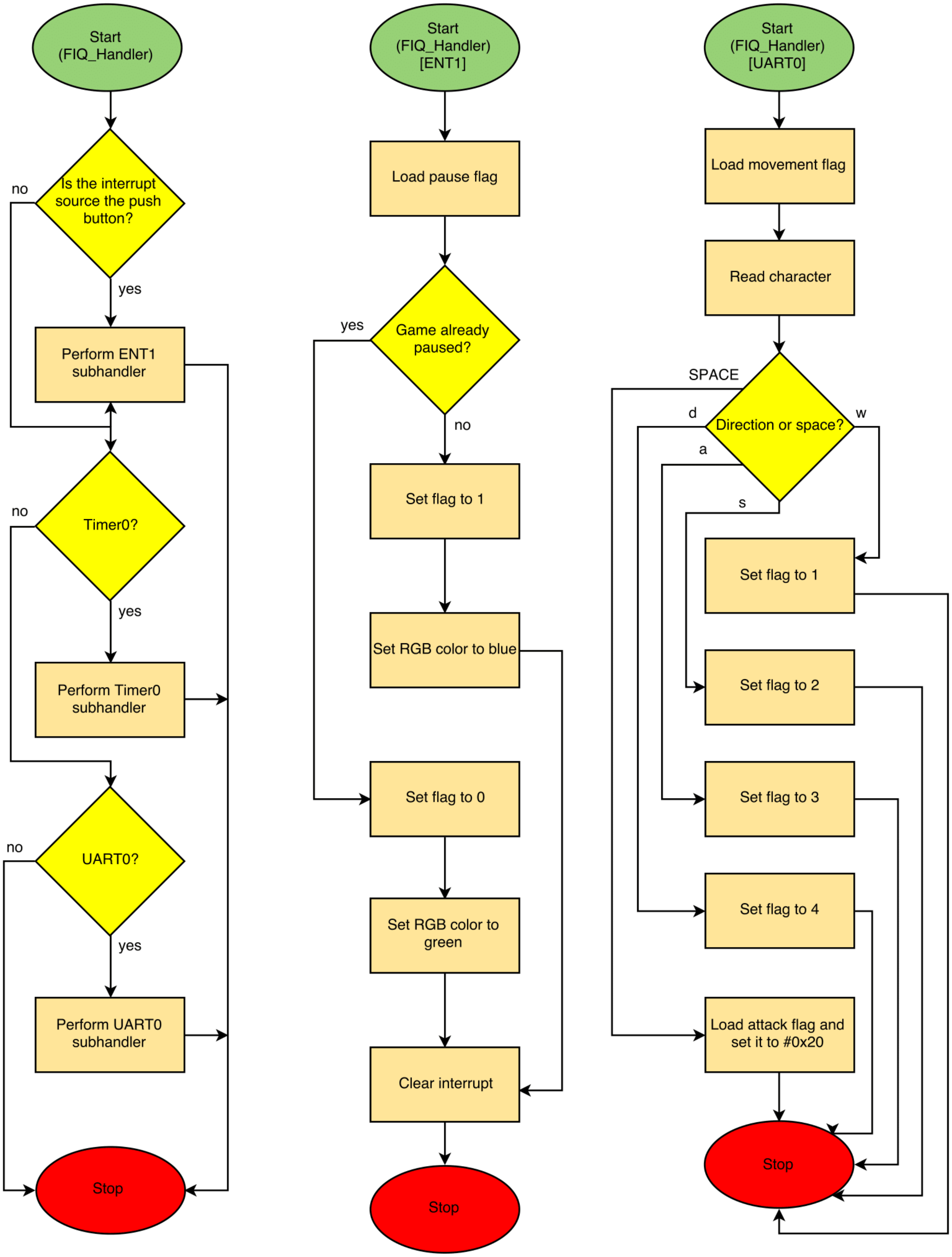

Start
(collision_adjust)
Are the coordinates equal?
yes
no
Reset user coordinates to starting, direction to the right
Load the number of lives left
Load the number of lives left
All lives used?
yes
no
Increment and store back in byte location
Illuminate the # of lives left on the LEDs
Set r5 to 1 as a flag
Set r5 to 0 as a flag
Stop


Start
(board_reset)
Load board address
Set first row as all Z's followed by the end line characters
Set the next two as all spaces bordered by Z's
The next 7 as dirt characters bordered by Z's
The same but with a '>' in the middle
The next 7 as dirt characters bordered by Z's
The last row as all Z's followed by the end line characters
Null terminate the string
Stop

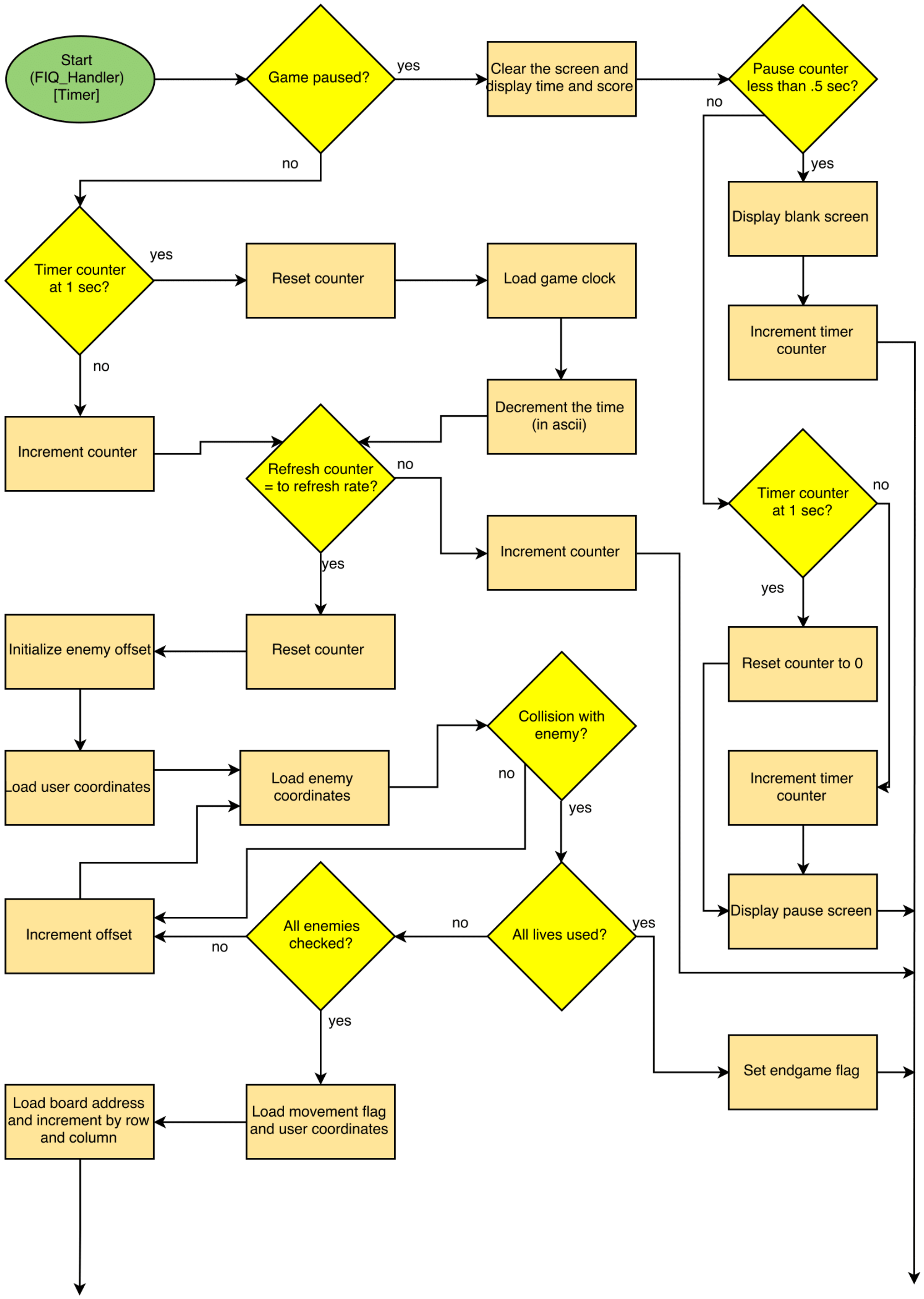

Start
(end_chars)
Store a new line character
Increment address
Store a line return character
Increment address
Stop

Wee Dig Dug

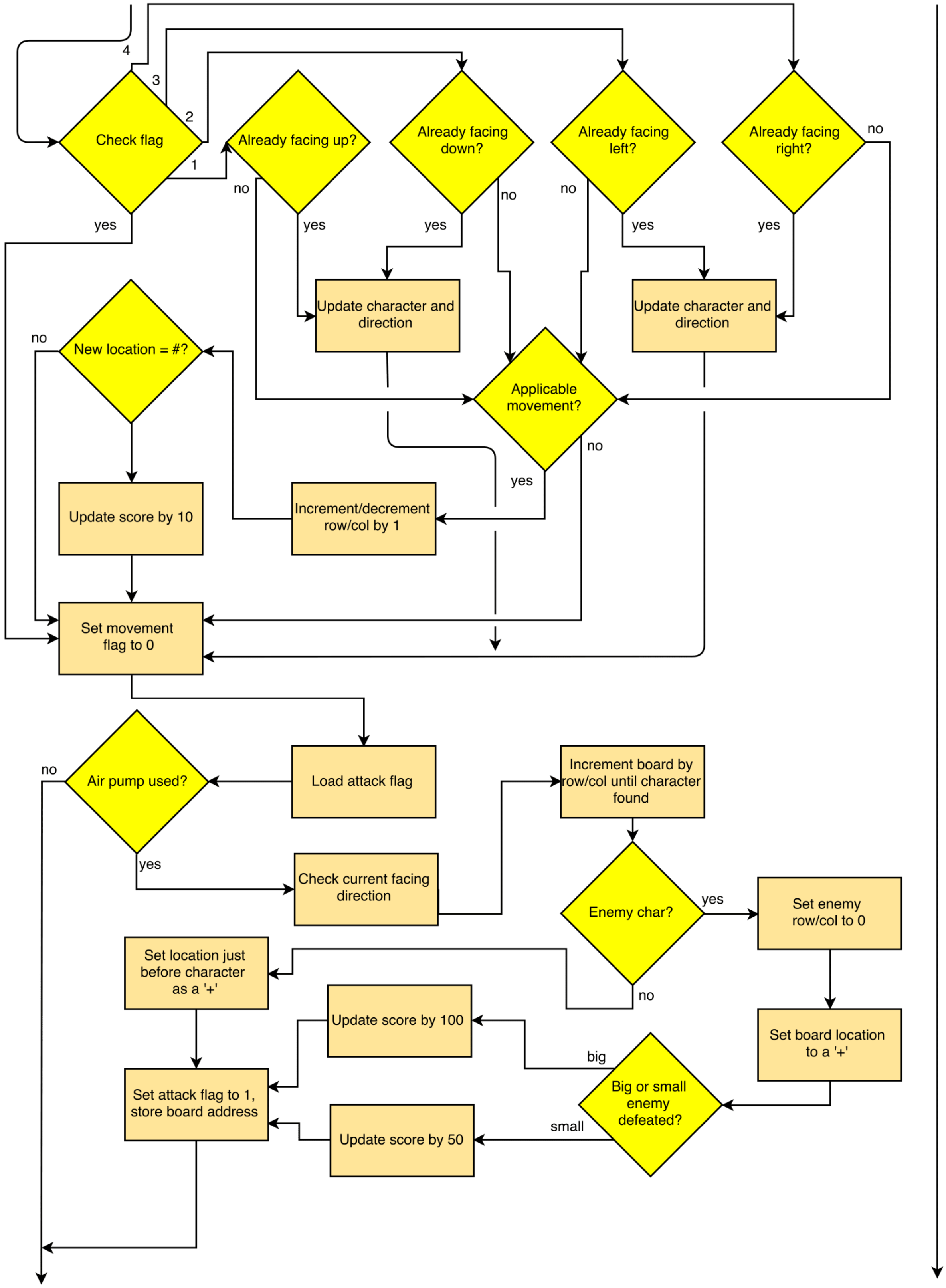

Start
(reinitialize)
Reset user coordinates to initial location and direction
Reset all enemy characteristics to initial
Reset all memory flags to 0
Reset the board
Stop
Start
(end_initialize)
Reset the time to 120 seconds
Reset the score to 0
Reset the lives left back to original
Revert the refresh rate back to .5 sec
Set the level to 1
Stop


Start
(FIQ_Handler)
Is the interrupt source the push button?
no
yes
Perform ENT1 subhandler
Timer0?
no
yes
Perform Timer0 subhandler
UART0?
no
yes
Perform UART0 subhandler
Stop
Start
(FIQ_Handler)
[ENT1]
Load pause flag
Game already paused?
yes
no
Set flag to 1
Set RGB color to blue
Set flag to 0
Set RGB color to green
Clear interrupt
Stop
Start
(FIQ_Handler)
[UART0]
Load movement flag
Read character
Direction or space?
SPACE
d
a
s
w
Set flag to 1
Set flag to 2
Set flag to 3
Set flag to 4
Load attack flag and set it to #0x20
Stop

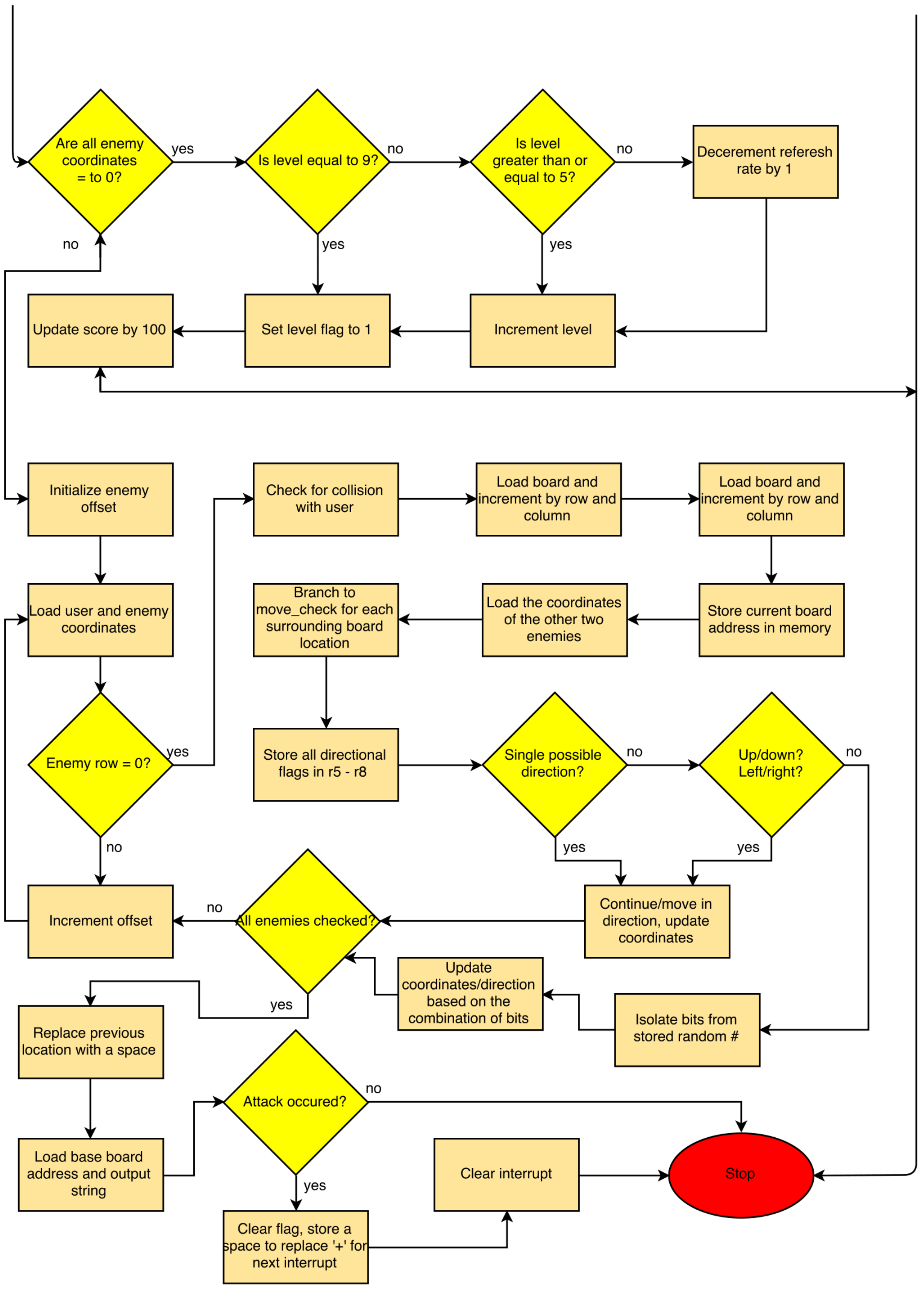
Wee Dig Dug


Start
(FIQ_Handler)
[Timer]
Game paused?
yes
no
Clear the screen and display time and score
Pause counter less than .5 sec?
no
yes
Display blank screen
Increment timer counter
Timer counter at 1 sec?
yes
no
Reset counter
Load game clock
Decrement the time (in ascii)
Increment counter
Refresh counter = to refresh rate?
no
yes
Increment counter
Timer counter at 1 sec?
no
yes
Reset counter
Initialize enemy offset
Reset counter to 0
Increment timer counter
Collision with enemy?
no
yes
Load user coordinates
Load enemy coordinates
Increment offset
All enemies checked?
no
yes
All lives used?
no
yes
Display pause screen
Set endgame flag
Load board address and increment by row and column
Load movement flag and user coordinates

Wee Dig Dug



4
3
2
1
Check flag
Already facing up?
Already facing down?
Already facing left?
Already facing right?
no
yes
no
yes
yes
no
no
yes
yes
Update character and direction
Update character and direction
no
New location = #?
Applicable movement?
no
yes
Update score by 10
Increment/decrement row/col by 1
Set movement flag to 0
Load attack flag
no
Air pump used?
yes
Check current facing direction
Increment board by row/col until character found
Enemy char?
yes
Set enemy row/col to 0
no
Set location just before character as a '+'
Update score by 100
Set board location to a '+'
big
Big or small enemy defeated?
Set attack flag to 1, store board address
small
Update score by 50


Are all enemy coordinates = to 0?
yes
no
Is level equal to 9?
no
yes
Is level greater than or equal to 5?
no
yes
Decerement referesh rate by 1
Update score by 100
Set level flag to 1
Increment level
Initialize enemy offset
Check for collision with user
Load board and increment by row and column
Load board and increment by row and column
Load user and enemy coordinates
Branch to move_check for each surrounding board location
Load the coordinates of the other two enemies
Store current board address in memory
Enemy row = 0?
yes
no
Store all directional flags in r5 - r8
Single possible direction?
no
yes
Up/down? Left/right?
no
yes
Increment offset
no
All enemies checked?
yes
Continue/move in direction, update coordinates
Update coordinates/direction based on the combination of bits
Isolate bits from stored random #
Replace previous location with a space
Attack occured?
no
yes
Load base board address and output string
Clear interrupt
Stop
Clear flag, store a space to replace '+' for next interrupt