Lab #6

CSE-379

Introduction to Microprocesssors

Ben Miller
bdm23

Andrew Zhou
amzhou

Lab Section: R2

May 1st, 2021

Contents

1 Division of Work

1.1 Partner 1 - Ben

Ben worked on the following:

- Properly displaying the board, which included...
  - Transcribing the boards into a lab6boards.s file, including the completed flows tally, the time, the X's that make up the border, and the seven basic pairs of endpoints
  - Writing a key for the lines to be drawn in the correct colors
  - Converting the ASCII characters into visible lines and ensuring the player never sees the underlying characters
- Player movement logic, which includes...
  - Basic character movements, allowing the player to move up, down, left, and right
  - Preventing the player from moving into a wall
  - Preventing the player from moving into the O already being drawn from
- Portions of the pause functionality, which included...
  - When the game is paused, the player is notified as such
  - When the game is paused, the current board is hidden
  - When the game is paused, the player is given the option to quit, restart, resume, or play on a new board.
- Writing the documentation LaTeX file

1.2 Partner 2 - Andrew

Andrew worked on the following:

- Basic line drawing logic, which includes...
  - When a player is moving but a line had not been previously selected, no line is drawn
  - The player can begin drawing a color by moving over an endpoint and hitting the space bar
  - Printing the different line types, those being '|', '-', and '+'
- The timer, which increments at a rate of once per second
- The number of flows being maintained at the top of the board
- Portions of the pause functionality, which included...
  - Preventing the timer from incrementing while the game is paused
  - Preventing player movement while the game is paused

  - Acting upon the action the player chooses whether the game is paused (i.e. if the player chose to quit, the game exits)

- Importing the RGB functionality from previous labs and adapting it for use in this assignment
- Making the flowsheets for the documentation

1.3 Cooperative Work

Throughout the development of the project, both team members were in voice calls debugging and coding together. In addition to this, the following portions of the project were completed cooperatively:

- Advanced line drawing logic, which includes...
  - Pressing the space bar while already drawing a line deletes the line currently being drawn
  - If a line is drawn for which that color has already been drawn, the previous line is deleted
  - Crossing over a different color's line deletes said line
  - The player cannot move over the current line being drawn
- Informing the player when the game is over from beating the game (not when the game is paused, this functionality was split up as previously mentioned)
- Randomly selecting the board
- Bug testing

2 Program Overview

2.1 How to Use the Program

- To compile and run the program, follow the following steps.
  1. Copy lab6.s, lab6library.s, lab6startup_code.c, lab6wrapper.s, and lab6boards.s to your local machine.
  2. Open these files in Code Composer Studio. Ensure that the program recognizes the ARM M4 Cortex.
  3. Open PuTTY and SSH into the ARM M4 Cortex.
  4. Build and run the project, then select "Resume." The program is now ready to be used.
- Once you select "Resume", a prompt will ask you to press the space bar. After doing so the game will begin, randomly selecting one of 16 different boards.
- You will be met with a 7x7 grid composed of tiles, with the borders designated by X's. The topmost leftmost tile will be highlighted; this highlighted tile will be referred to as the "player". 14 other tiles will house colored circles; these will be referred to as "dots".
- At the top of the board will be two pieces of useful information: the number of pairs, which will be revisited shortly, and a timer. The timer increments once per second and represents the time elapsed on the current board.
- The goal of the game is to connect the seven pairs of dots to one another. A "flow", or a line between two dots, can be made by highlighting a single dot and pressing the space bar. Then, move your way to the corresponding color to complete the flow. For every completed flow, the completed value at the top of the board will increment by one.
- Player movement is simple: press the WASD keys to respectively move up, left, down, and right. The player cannot move on to or top of a wall. Accordingly, this means the player cannot leave the 7x7 grid.
- As previously mentioned, a player begins drawing a flow by pressing the space bar over a dot. The flow is composed of three different types of lines. A pipe, '|', represents vertical movement. A dash, '-', represents horizontal movement. A corner, '+', represents a change in direction from vertical to horizontal or horizontal to vertical.
- There are rules that must be made aware of while a line is being drawn:
  - The player cannot draw through the flow currently being drawn.
  - If the player is drawing a flow, they cannot enter a mismatching dot to end the flow.
  - The player cannot end a flow at the dot at which the flow began.
  - If a flow is drawn through a flow of a different color, the previous flow is completely deleted.

  - If the color is selected for a flow that has already been drawn, the previous line is completely deleted.
  - At any time while drawing a flow, the player can press the space bar to delete the line currently being drawn.

- The game can be paused by pressing SW1 on the ARM Cortex M4. Multiple actions can be performed while the game is paused.
  - The user can press 'q' to immediately and completely exit the game. Note that the user can press 'q' at any moment to quit the game, even if the game isn't currently paused.
  - The user can press 'r' to resume the current board exactly where they left off. If a line was being drawn before the game was paused, the line will continue being drawn after the game resumes. At the bare minimum, the cursor will be exactly where the user left it.
  - The user can press 't' to restart the current board. The flow counter and timer will both be reset to zero, and any flows will be deleted.
  - The user can press 'g' to start a new game with a new board.
- The RGB LED on the ARM Cortex M4 will light up to represent the color currently being drawn. If no color is being drawn, the LED is off.
- Once the game is completed (i.e. all seven pairs of dots have been matched) a congratulations screen will appear. From here, the player can choose to restart with a new board (by pressing 'g') or exit the game (by pressing 'q').

2.2 Program Summary

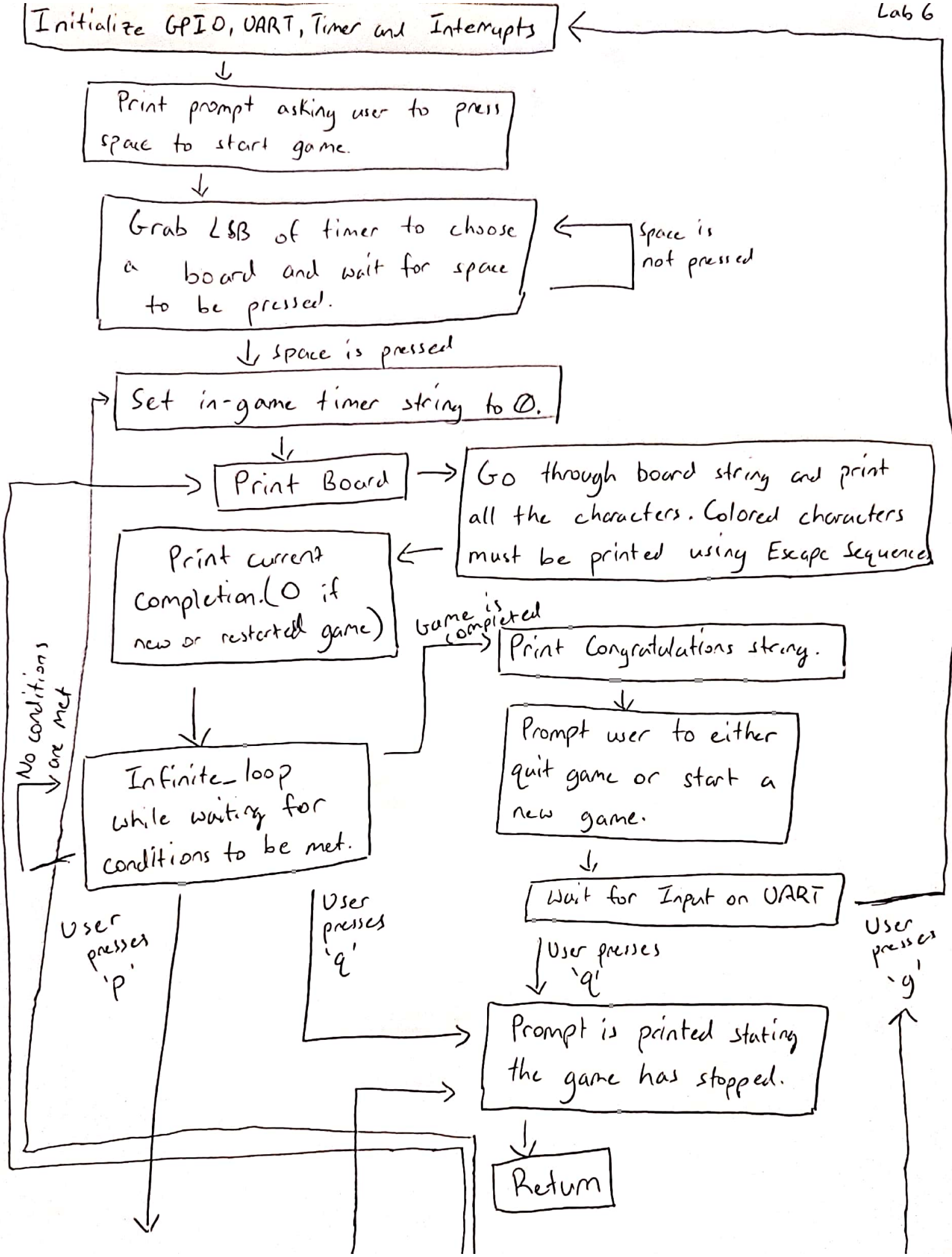
This subsection will describe the main `lab6` subroutine. The overwhelming majority of the program is contained in the handlers, however the functionality inside `lab6` is still important.

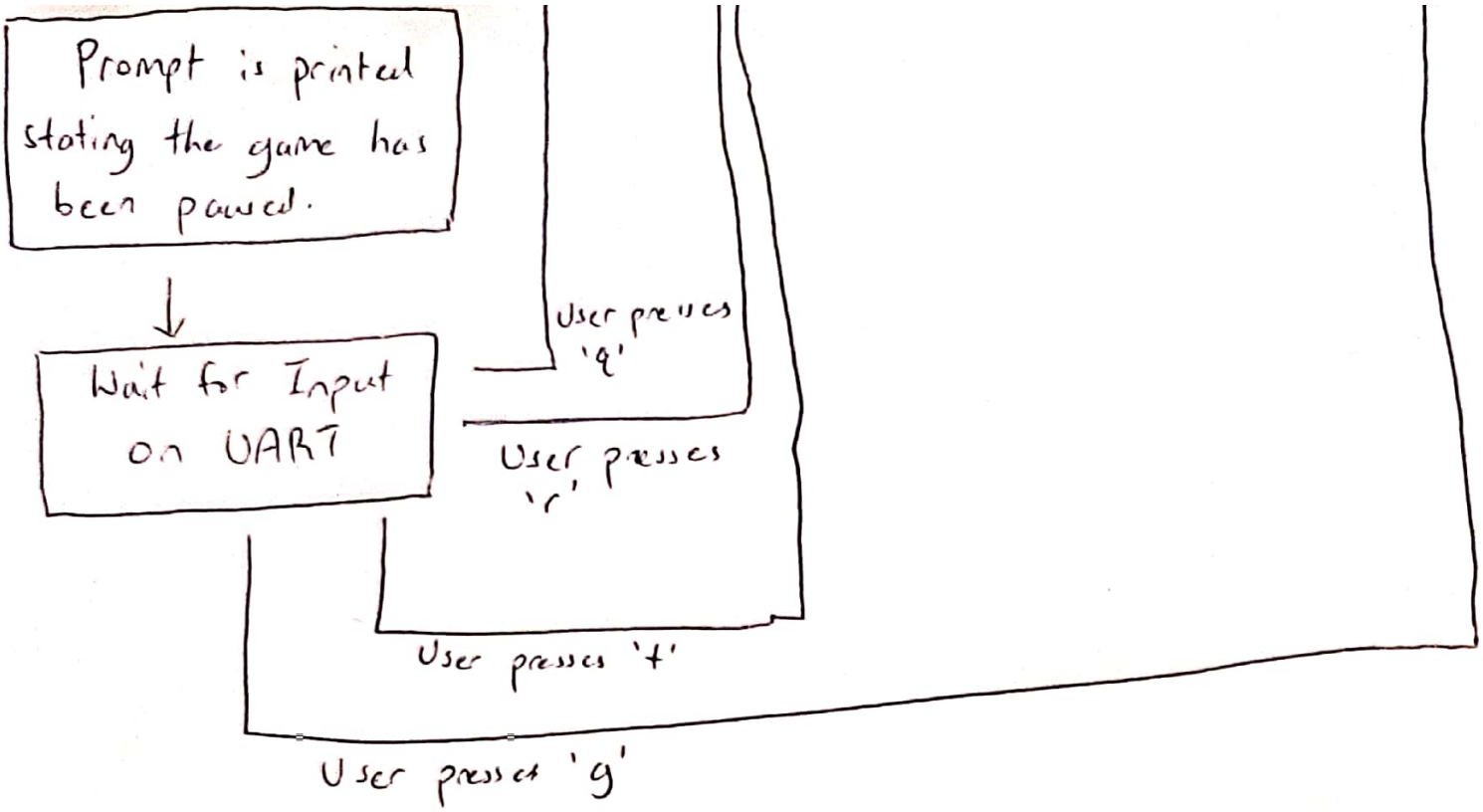
- The `lab6` subroutine itself is fairly short, only storing the r0-r12 and the lr register on the stack.
- `lab6` immediately steps into the `new_game` subroutine, which initializes the handlers.
- After the handlers are initialized, `new_timer_init` is called, which initializes the timer to tick once every sixteenth of a second.
- The user is prompted to press the space bar. Upon doing so, the timer uses a value in the `ptr_to_choose_board` to select a pseudorandom board number.
- With the board number now discovered the program enters `start_game_loop`. This subroutine initially loops indefinitely until the player presses the space bar indicating they wish to begin the game.

- After the space bar is pressed, the board number is used to print the board this game will take place on. Initially, this is done through a subroutine called `print_board_og`, although there is another subroutine called `print_board_edit` that works almost identically. The latter will be covered in more detail later.
- For now, the program resides in `print_board_og`, which uses the number returned by `new_timer_init` to load the correct pointer to the board. Once this pointer is loaded, the program branches to `inital_board_printed`.
- This function mainly acts as a bridge between `print_board_og/edit` and `resume_board_print`. The one important thing done here is a comparison that checks if the board is being resumed, in which case the timer is maintained. If this comparison fails (indicating this being a new game or a game that was restarted) the timer is initialized to 0000.
- The program is now in `resume_board_print`, a simple subroutine that just outputs the correct board. However, this board needs to be "ANSI corrected."
- The boards are stored as strings using ASCII characters to represent the different colors of circles, pipes, etc. (for a key, see the first section of the appendix). Of course, the user shouldn't be expected to have a key on hand to decipher the game board, so these ASCII values must be converted into something readable.
- This is done by a combination of multiple subroutines. First, after `resume_board_print` the program enters `print_char`. This subroutine determines what the current tile's ASCII value is, whether it be a blue circle or a white corner.
- This information is used to call a helper function specifically designed for each ASCII character. For example, if the ASCII is a green pipe, `print_green_pipe` is called.
- These helper functions convert the tile from an ASCII value to a colored ANSI escape sequence This is done for every tile on the board using a for loop.
- Once this is complete, the program enters `chars_done`, which moves the cursor to the starting position and internally resets the current x and y coordinates.
- If the game was previously resumed, then the game restores the previous x and y coordinates both visually to the user and internally. This is specifically done with `restore_x` and `restore_y`.
- The board now enters `print_completion`, which is called whenever a flow is completed. This value is initialized to either 0 or whatever it previously was, in the event the game was resumed.
- After all of this, the board has finally been properly printed and the program enters `infinite_loop`, where much of the runtime is spent inside.

- `infinite_loop` checks for a couple of important things. Firstly, if an internal value inside of the `ptr_to_choose_board` string is set to 1, the board is re-printed. The reasoning behind this will be explained later inside the UART handler.
- Next, the loop checks to see if the game has been completed or if the user has quit. If either of these are true, they branch to `completed_game` and `quit_game`, respectively.
- The former alerts the user that the game has been completed and gives the user the option to either start a new game or exit.
- The latter occurs when the user has opted to quit the game during play, resulting in the program exiting.
- If neither of these things are true but the game isn't paused, this loop restarts. This occurs indefinitely until the game is paused, after which `pause_loop` is entered.
- This subroutine is the final major component of the `lab6` subroutine. This is a while loop that indefinitely loops upon itself until one of four buttons are pressed.
    1. If a 'q' was pressed, the game exits.
    2. If a 'r' was pressed, the game resumes its previous board and undergoes all of the steps noted above for a resume.
    3. If a 't' was pressed, the game restarts the current board, clearing it of any edits.
    4. Finally, if a 'g' was pressed, the user is given a new board. The previous board gets deleted of its edits before the new board is printed. For both 't' and 'g', this is done with the `delete_edit` subroutine.

2.3 Flowsheet of the Entire Program


Initialize GPIO, UART, Timer and Interrupts
Print prompt asking user to press space to start game.
Grab LSB of timer to choose a board and wait for space to be pressed.
Space is not pressed
Space is pressed
Set in-game timer string to 0.
Print Board
Go through board string and print all the characters. Colored characters must be printed using Escape Sequence
Print current completion. (0 if new or restorted game)
Game is completed
Print Congratulations string.
Prompt user to either quit game or start a new game.
No conditions are met
Infinite_loop while waiting for conditions to be met.
Wait for Input on UART
User presses 'p'
User presses 'q'
User presses 'q'
User presses 'g'
Prompt is printed stating the game has stopped.
Return


Prompt is printed stating the game has been paused.
Wait for Input on UART
User presses 'q'
User presses 'r'
User presses 't'
User presses 'g'

3 Subroutine Descriptions

3.1 UART_Handler

The functionality of the UART handler is by far the most complex component of the program. By default, the UART handler checks for two conditions: whether the game is paused or unpaused. This section will be appropriately divided based on this condition.

3.1.1 Paused

The functionality while the game is paused is comparatively simple.

- This portion of the code is entered when the internal `ptr_to_pause` string indicates that the game is currently paused.
- Should this happen, the UART handler branches into `ptr_to_pause_conditions`. Inside of this subroutine, the game takes note of what button was just pressed, whether that be 'r', 't', 'g', or 'q'.
- If none of these buttons are pressed, the handler recognizes this as an invalid input and exits.
- This information is passed into `uart_store_char`, which stores this back into `ptr_to_pause` and then exits.
- The value stored is utilized by `pause_loop` (see the program summary section) to take the action corresponding to the button the user pressed.

3.1.2 Unpaused

- Should the internal pointer indicate the game is not in fact paused, the program instead branches into the `uart_game` subroutine.
- Almost everything from here on in some way modifies the board, so the board to be edited must be found. The board number is extracted from an internal string and the pointer to the board we wish to edit is stored in a register.
- Once the board is found, the program branches into `uart_board_found`. The color being drawn is found using the `ptr_to_choose_board` string, which also contains the board number.
- Next, the player's current location on the board is found. Inside a string called `ptr_to_info` is the current x and y coordinates of the player.
- These are loaded. Then, the below formula is used to calculate the current index on the grid as a value that can be added to a pointer.

$$(\text{colindex} * \text{rowwidth}) + \text{rowindex} \tag{1}$$

- With all of this being done, the program now checks what key was pressed. The handler checks for three things: if the button was movement via WASD, if the button was a 'q', meaning the user desires to quit the game, or if the button was a space bar, indicating a desire to begin drawing a flow.

- If the program concludes none of these were the button that was pressed, an invalid button was pressed and the handler exits.
- The functionality of each of these is extensive, therefore each will receive their own subsection.

Normal Movement
The functionality of the movement is practically identical between up, down, left, and right. The differences are limited to small changes in pointer math, and because of their similarities they will all be simultaneously described.

- The first thing the movement handler does is discern if the movement itself is valid. Here, this is done by adding/subtracting a value to ensure the player is not attempting to move into a wall. If this is the case the program exits.
- Next, the program checks if the player is attempting to move in to an O. This functionality is a large edge case in and of itself and will receive its own section below. This section will purely cover normal movement.
- If the player isn't attempting to move into an O, the program immediately skips to `up/down/left/right_allow_move`.
  - *Note: For conciseness, up/down/left/right will hereafter be simply referred to as "direction." For example,* ***`up/down/left/right_allow_move`*** *would be called* ***`direction_allow_move`****.*
- The first thing this subroutine does is calculate the ASCII value of the tile the player wishes to move in to. This is done utilizing equation 1, described above. This ASCII value is saved.
- The program next ensures that the player isn't attempting to write into itself, a conclusion made by determining that the ASCII value saved was not of the same color as the color currently being drawn. For example, if the color being drawn is yellow and the player is attempting to move into an already existing yellow corner, the movement must be blocked.
- If the movement is illegal, the program exits. If no color is being drawn, this above step is skipped in its entirety.
- If the program hasn't exited yet, the movement is legal. The movement is allowed to be executed and the new x/y-coordinate is stored internally. If no flow is currently being drawn, the handler exits right now.
- Otherwise, a flow is currently being drawn, and there are more pieces of logic that must be checked. The first piece of logic is if a different colored line is being drawn over.
- This is discerned by checking that the ASCII value being moved in to is between 0x3F and 0x53, which are a red vertical pipe and a white corner, respectively. If the ASCII value is between these two, the program understands that the user is drawing over a line.

- ASCII arithmetic is utilized to check whether the player is moving into a pipe, dash, or corner. From here, the program is able to determine what color the player is moving in to, after which all non-dot values of that color are deleted from the board.
- Should this happen, the program sets an internal value inside of `ptr_to_choose_board` to 1. This is used by the infinite loop to recognize that the board needs to be printed. The number of completed flows is also reduced by 1.
- Once any existing conflicts are resolved, the movement is printed. If the player is moving up or down, a pipe is printed. If the player is moving left or right, a dash is printed.
- This change is inserted into the board string currently being edited, as the player's movement must be both internal and external.
- The last piece of logic is whether or not the player just changed directions, implying that a corner needs to be printed. This is done last, as more often then not the player is not changing directions.
- The first step to this calculating if the player did indeed change directions. Using the color of the line being drawn which was previously stored, ASCII arithmetic is used to assign a register with the value corresponding to the colors pipe (if a dash was just drawn) or dash (if a pipe was just drawn).
- This register is compared with tile the player was previously on. For example, if the player had just moved up, the value underneath the player's current location is the previous location and that ASCII value is stored.
- The previous location's ASCII value is compared to the opposing line (i.e. if the player had just moved up, the previous location is checked to see if it was a horizontal dash). If this comparison is false, the player didn't change directions and the program exits.
- If the comparison was true, the previous value is changed into a corner of the correct color, after which the program exits.

Movement into an O

While a majority of the logic behind movement into an O is identical to normal movement, there are key differences between how the two are handled that are worth mentioning.

- If the program did indeed conclude the player is attempting to move into an O, a subroutine called `direction_has_O` is entered.
- This subroutine must first determine if the player is attempting to move in to the original O that this flow started at. This is done by comparing the coordinates of said original O, stored in `ptr_to_first_o`. If the coordinates match, the program exits.
- If the program didn't exit, that means that this isn't the original O. The color between the two Os must match. If there is currently no line being drawn, implying there is no color to match, the program jumps straight to `direction_allow_move`.

- If a line *is* being drawn and the colors *do not* match, the handler exits. If they do match, the handler jumps to `direction_clear`.
- If the program does indeed make it inside `direction_clear`, the flow being drawn is being completed. The LED on the ARM M4 Cortex is cleared. Next, the flow tally is incremented by 1. Finally, the pointer to the first o is reset to allow for future movements.
- From here, the program steps into `direction_allow_move`, which for the most part works identically to normal movement. The only deviation is if the user is completing a flow. Should this be the case, `direction_corner` is entered.
- This subroutine resolves an edge case where a user could not properly draw a corner into an O to complete a flow.
- This subroutine resolves this edge case through a change in logic. With the normal movement, the O will assume any movement in to it is a corner because a pipe/dash isn't equal to an O. A slight change in code resolves the problem, hence the need for this subroutine.
- The movement has now been completed, into an O or not, and the handler exits.

Quitting The next piece of handler functionality is a simple one. If the button pressed was a 'q', an internal string is modified to inform the loop that the user has quit, and the handler immediately exits.

Space Bar
The final major piece of logic behind the UART handler is the if a user pressed a space bar.

- The first thing the handler must do when the space bar is pressed is grab the board and save it. The integer 40 gets added to the now saved pointer to direct it towards the first valid tile on the grid.
- Next, the program determines whether or not this is the first space press (the user wants to draw a line) or the second space press (the user wants to delete a line currently being drawn).
- This is done by comparing the color of the line being drawn to 0. If the color is 0, a line isn't being drawn and this is the first space bar press. Otherwise, it is the second space bar press and the program branches to `second_space_delete`. This will be explained later.
- If this is the first space press, we must check that we are currently hovering over an O. If this isn't an O, the space bar press does nothing.
- Using equation 1, the ASCII value of the current tile is found. If the value is an O, the program branches into `uart_space_o`. Otherwise, the handler exits.

- If we are inside `uart_space_o`, that means the user is attempting to draw a new line. The color of the O being hovered over is stored and the proper LED is lit up.
- Then, the tiles up, down, left, and right of the O are checked. If any of them is a previous line that was drawn, the line is deleted by branching into `overwrite_line`. This will also be explained later.
- If none of the tiles are problematic, then the handler exits with its now lit up LED and after storing the values of this first O into `ptr_to_first_o`.
- If a previous line needs to be deleted, the program branches into `overwrite_line`. This function calls a subroutine called `second_space` and decrements the completed flow tally by 1, after which the handler exits.
- If this is the second space bar press, meaning the user wants to delete the line currently being drawn, the program branches into `second_space_delete`. This subroutine calls `second_space`, resets the LED, and resets internal strings to signify that a line is no longer being drawn. After all of this, the program exits.
- `second_space` is a function used by multiple portions of the UART handler. The subroutine itself is a for loop that accepts four arguments: a variable to be incremented as a counter and the pipe, dash, and corner of the color to be deleted.
- `second_space` goes through every tile in the program and compares it to one of these three lines given variables. If any of the comparisons are true, that specific tile is replaced with a space by a call to `delete_write`.
- Once the entire grid has been searched, `second_space` returns to its caller.

3.2 Switch_Handler

The functionality of the switch handler is perhaps the most simplistic portion of the entire program. This handler simply edits an internal string called `ptr_to_pause` which the `infinite_loop` utilizes to determine that the game is currently paused. After this singular action, this handler exits.

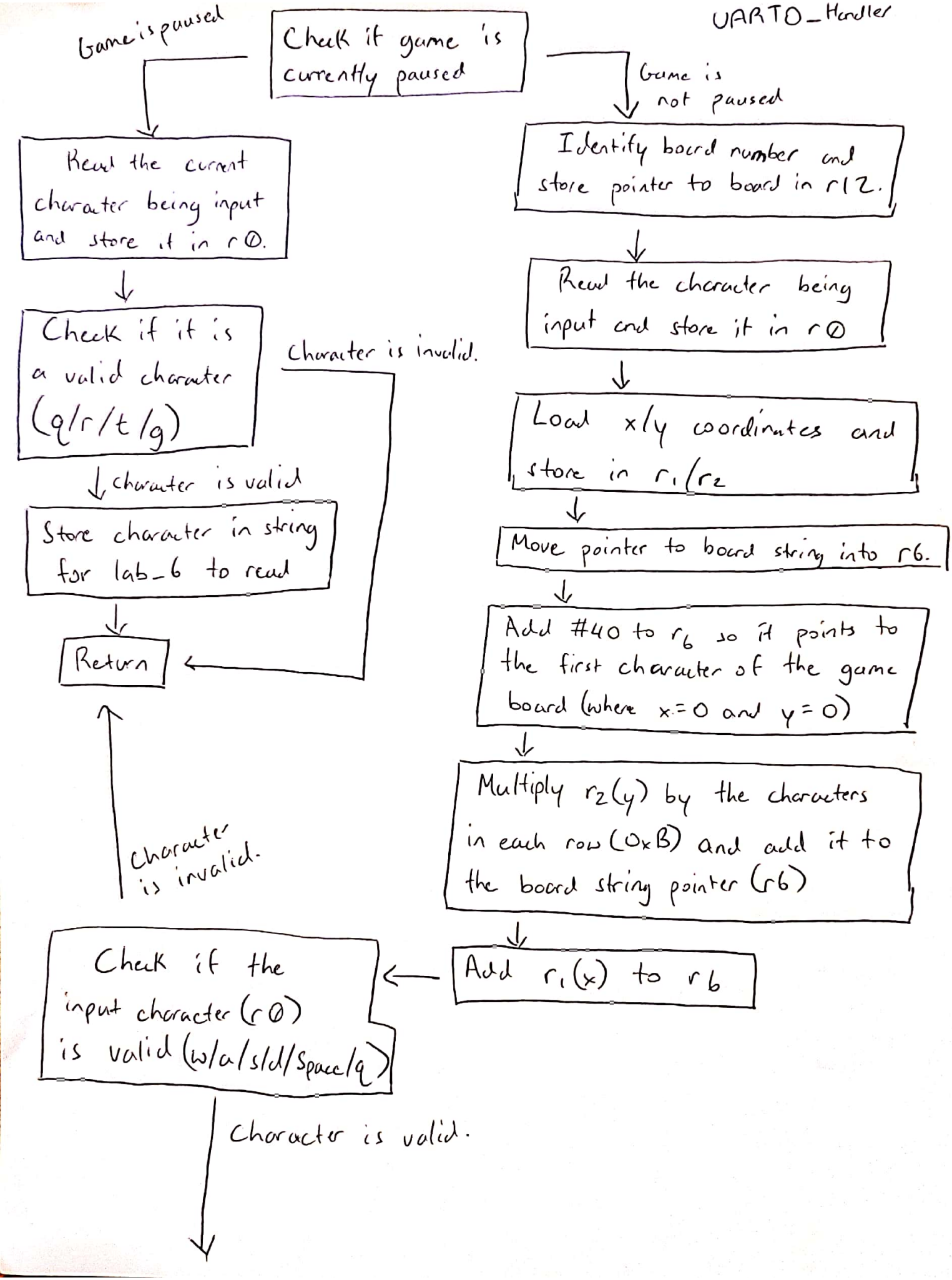
3.3 Timer_Handler

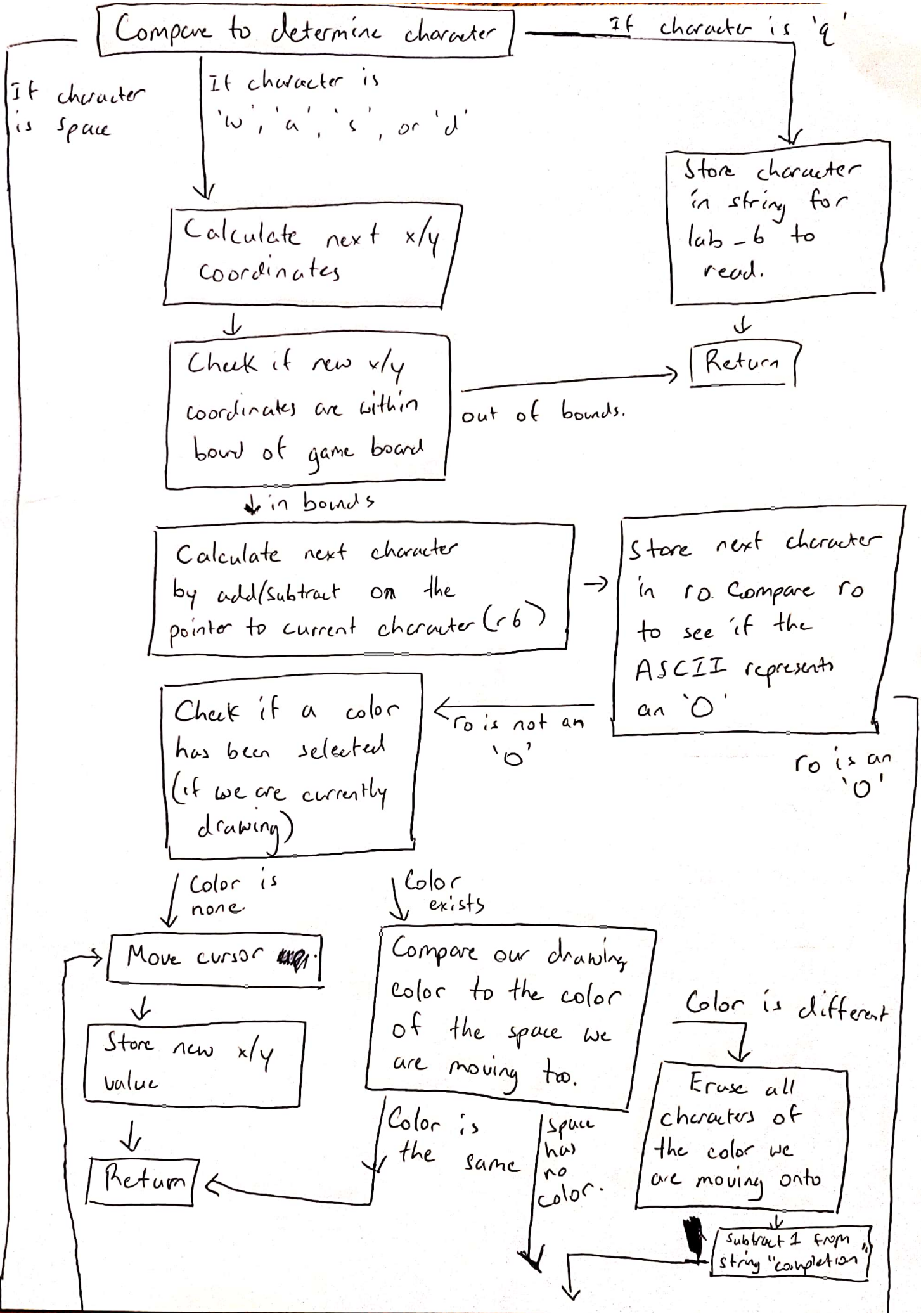
- This handler is used to increment the timer at the top of the board. It "ticks" once every sixteenth of a second, meaning the entire handler runs sixteen times a second.
- To start, `ptr_to_choose_board` is loaded and its value is compared to 0xF. If the value is indeed 0xF, 16 ticks have passed and the timer itself needs to be incremented.
- If the comparison is true, the handler branches into `timer_every_second`. If the comparison is false, the value is incremented and stored back into the pointer, followed by the handler exiting.

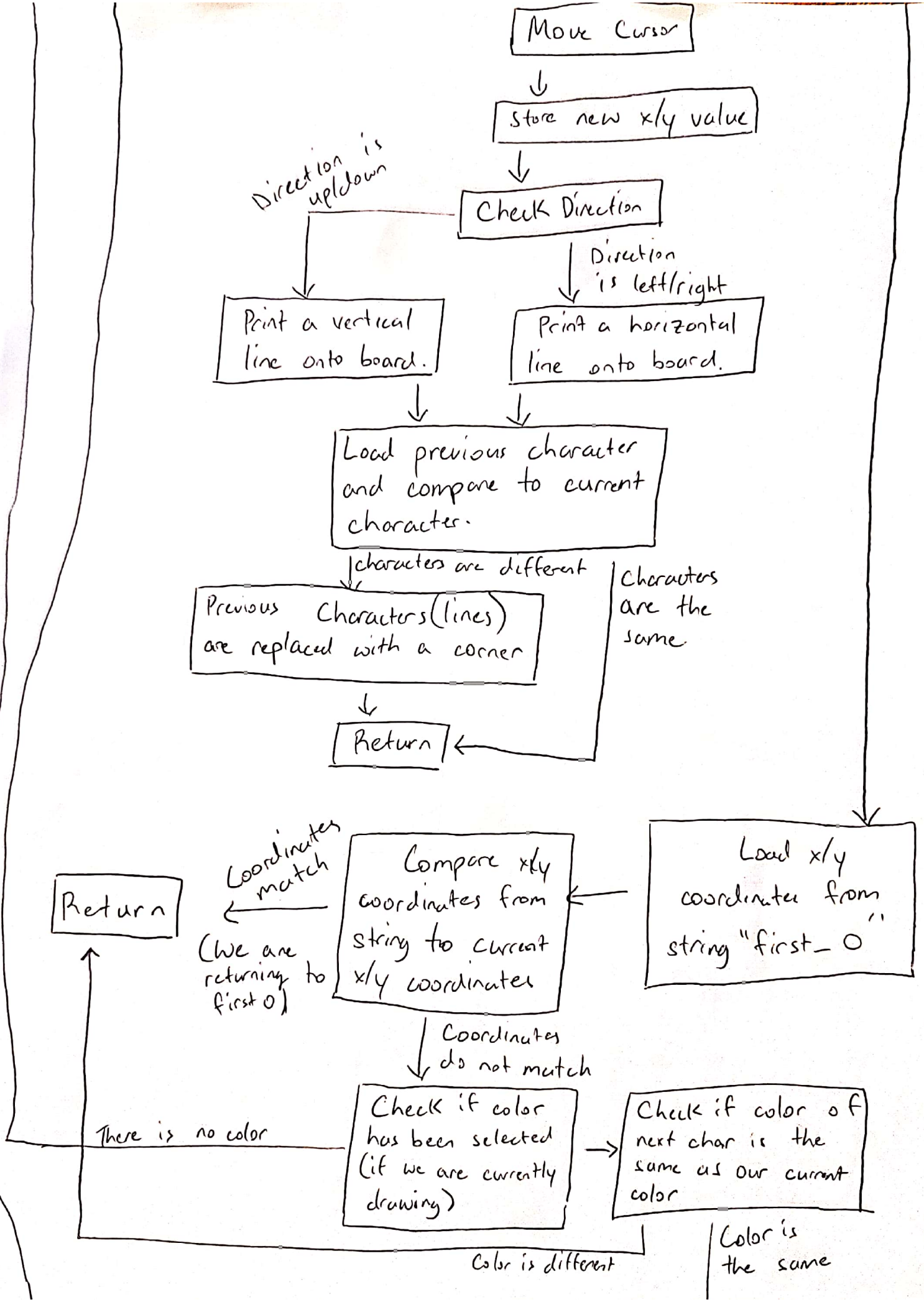
- This behavior is abused to pseudo-randomly select a board for the user. The timer is ticking every sixteenth of a second. At the beginning of the program, after the user presses the space bar, this hex value is converted to a board number and saved. From that point onwards, this value is simply used for incrementing the timer.
  - In other words, prior to the space bar being pressed at the very beginning of the game the handler is randomly selecting a board. After the space bar is pressed the timer is incrementing the handler counting on the board.
- If the value was indeed 0xF, the value from the pointer is set back to 0 and stored. Next, the handler checks to make sure the game isn't paused. If the game is paused, so is the timer. If the game isn't, the game is continuing as normal and the program steps into `timer_ones`.
- `timer_ones`, `timer_tens`, `timer_hundreds`, and `timer_thousands` are used to increment the timer on the board. They work in a sequential manner.
- In other words, they compare the maximum value of their slot to increment the timer.
  - `timer_ones` increments the ones slot of the timer. When this value becomes 9 (a time of 9), `timer_ones` sets it to 0 (a time of 10) and calls `timer_tens`.
  - `timer_tens` increments the tens slot of the timer. When this value becomes 9 (a time of 99), `timer_tens` sets it to 0 (a time of 100) and calls `timer_hundreds`.
  - `timer_hundreds` increments the hundreds slot of the timer. When this value becomes 9 (a time of 999), `timer_hundreds` sets it to 0 (a time of 1000) and calls `timer_thousands`.
- The time itself is printed through `timer_print`, a subroutine called by each of the above functions. Once the timer has been properly printed, the handler exits.

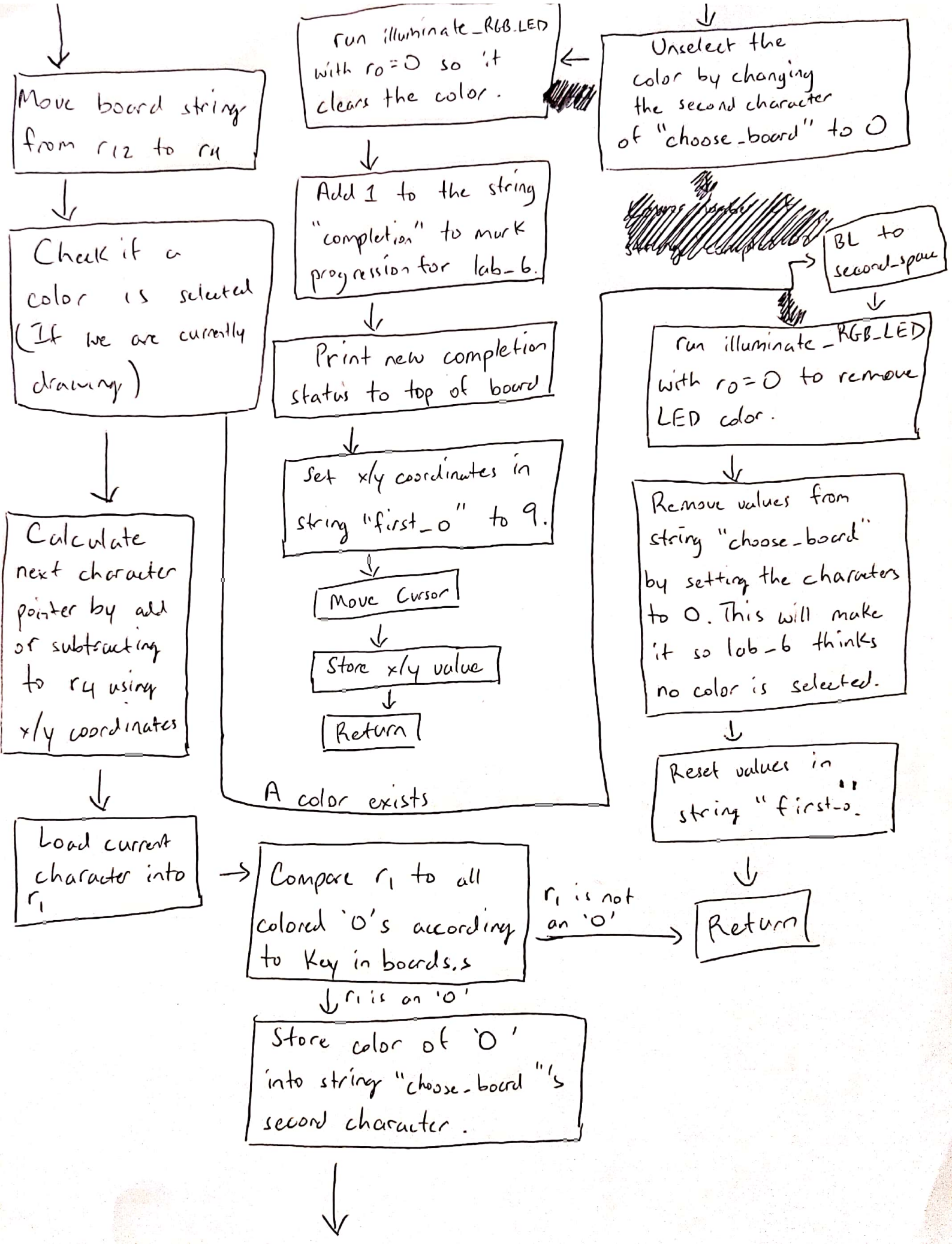
4 Additional Flowsheets

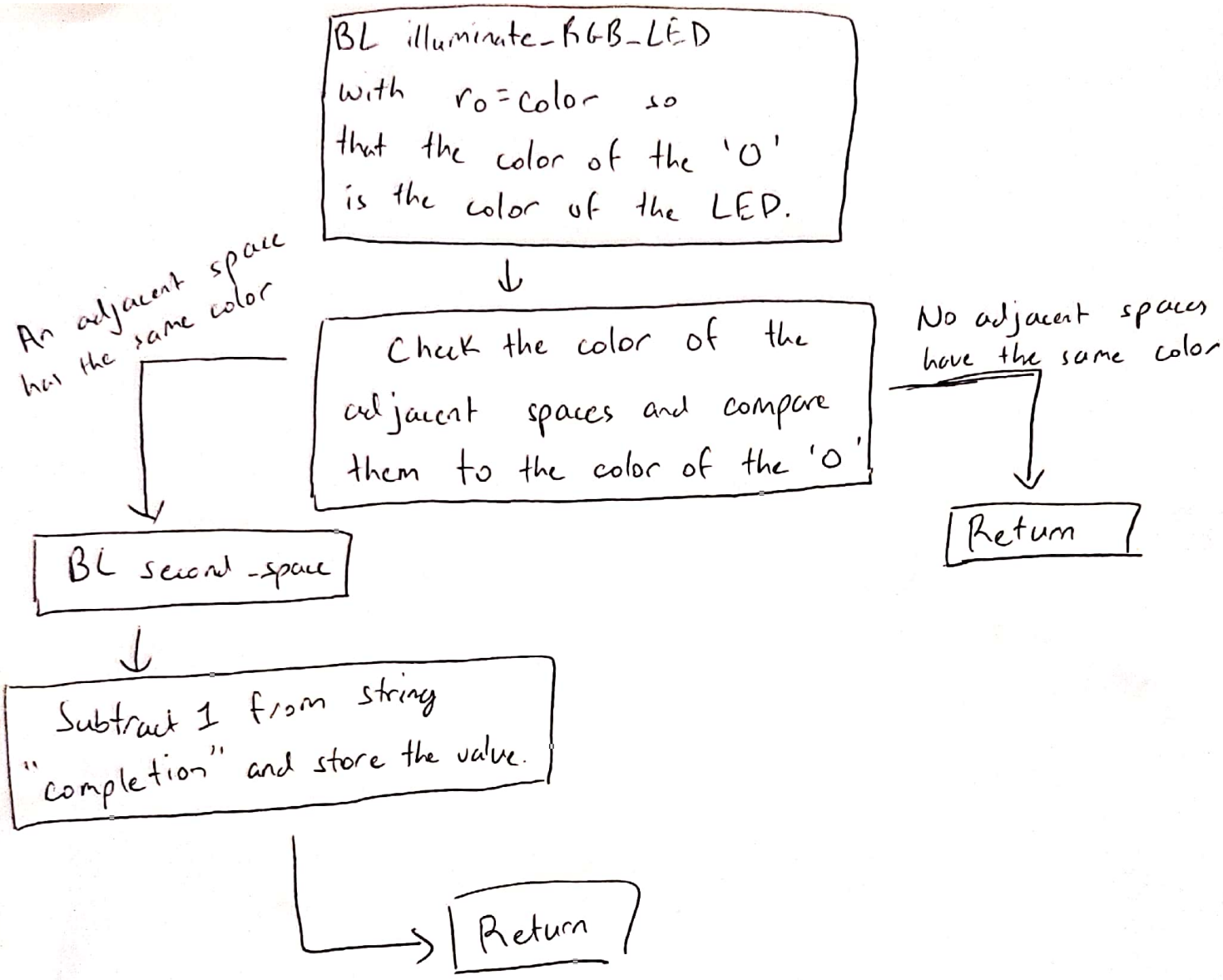
4.1 UART_Handler


UART0_Handler
Check if game is currently paused
Game is paused
Game is not paused
Read the current character being input and store it in r0.
Check if it is a valid character (q/r/t/g)
Character is invalid.
Character is valid
Store character in string for lab_6 to read
Return
Character is invalid.
Identify board number and store pointer to board in r12.
Read the character being input and store it in r0
Load x/y coordinates and store in r1/r2
Move pointer to board string into r6.
Add #40 to r6 so it points to the first character of the game board (where x=0 and y=0)
Multiply r2(y) by the characters in each row (0xB) and add it to the board string pointer (r6)
Add r1(x) to r6
Check if the input character (r0) is valid (w/a/s/d/Space/q)
Character is valid.


Compare to determine character
If character is 'q'
If character is space
If character is 'w', 'a', 's', or 'd'
Store character in string for lab_6 to read.
Calculate next x/y coordinates
Check if new x/y coordinates are within bound of game board
Return
out of bounds.
in bounds
Calculate next character by add/subtract on the pointer to current character (r6)
Store next character in r0. Compare r0 to see if the ASCII represents an '0'
r0 is not an '0'
r0 is an '0'
Check if a color has been selected (if we are currently drawing)
Color is none
Color exists
Move cursor
Store new x/y value
Return
Compare our drawing color to the color of the space we are moving to.
Color is the same
Space has no color.
Color is different
Erase all characters of the color we are moving onto
Subtract 1 from string "completion"


Move Cursor
Store new x/y value
Check Direction
Direction is up/down
Direction is left/right
Print a vertical line onto board.
Print a horizontal line onto board.
Load previous character and compare to current character.
characters are different
Characters are the same
Previous Characters (lines) are replaced with a corner
Return
Load x/y coordinates from string "first_0"
Compare x/y coordinates from string to current x/y coordinates
Coordinates match
(We are returning to first 0)
Return
Coordinates do not match
Check if color has been selected (if we are currently drawing)
There is no color
Check if color of next char is the same as our current color
Color is different
Color is the same


Move board string from r12 to r4
Check if a color is selected (If we are currently drawing)
Calculate next character pointer by add or subtracting to r4 using x/y coordinates
Load current character into r1
Compare r1 to all colored 'O's according to Key in boards.s
r1 is not an 'O'
Return
r1 is an 'O'
Store color of 'O' into string "choose_board"'s second character.
A color exists
run illuminate_RGB_LED with r0=0 so it clears the color.
Add 1 to the string "completion" to mark progression for lab_6.
Print new completion status to top of board
Set x/y coordinates in string "first_0" to 9.
Move Cursor
Store x/y value
Return
Unselect the color by changing the second character of "choose_board" to O
BL to second_space
run illuminate_RGB_LED with r0=0 to remove LED color.
Remove values from string "choose_board" by setting the characters to O. This will make it so lab_6 thinks no color is selected.
Reset values in string "first_0".
Return


BL illuminate_RGB_LED
with r0=color so
that the color of the '0'
is the color of the LED.
Check the color of the
adjacent spaces and compare
them to the color of the '0'
An adjacent space
has the same color
No adjacent spaces
have the same color
BL second_space
Subtract 1 from string
"completion" and store the value.
Return
Return

second_space

This function is used to clear the board of all vertical, horizontal and corner lines of a specific color.

Arguments:

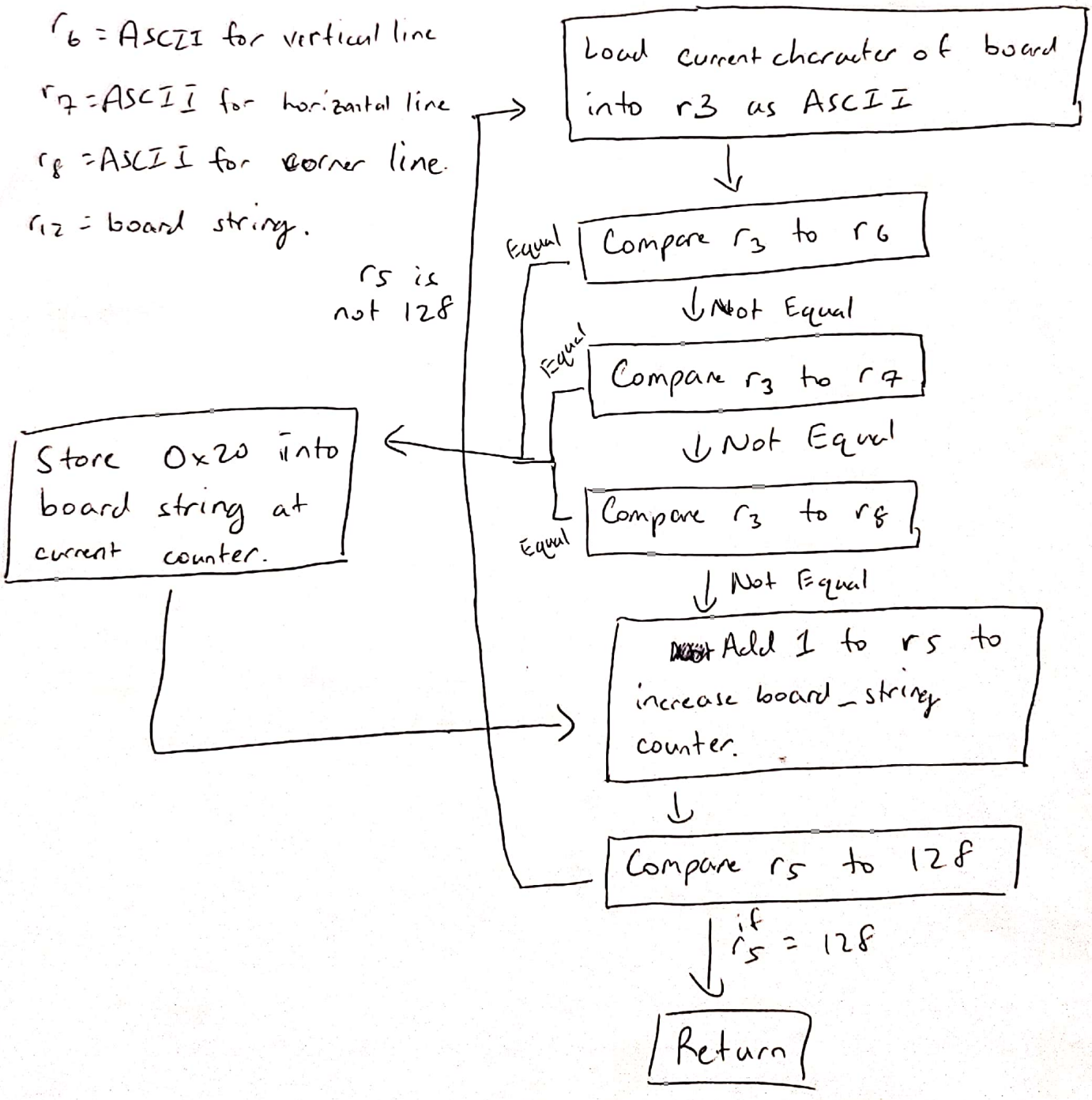
$r_5 = 0$

r_6 = ASCII for vertical line

r_7 = ASCII for horizontal line

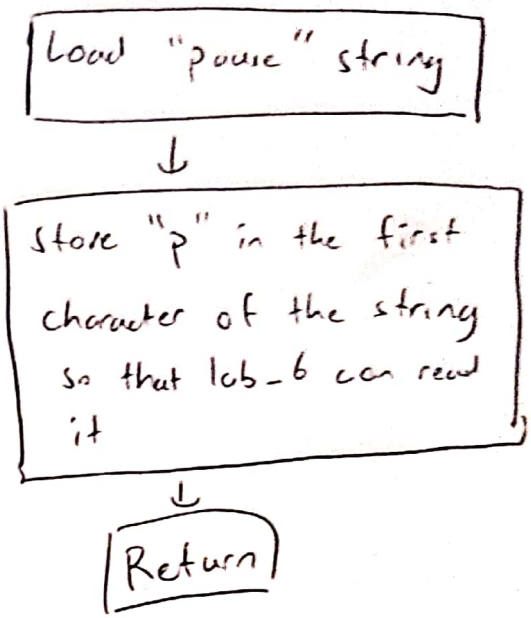
r_8 = ASCII for corner line.

r_{12} = board string.



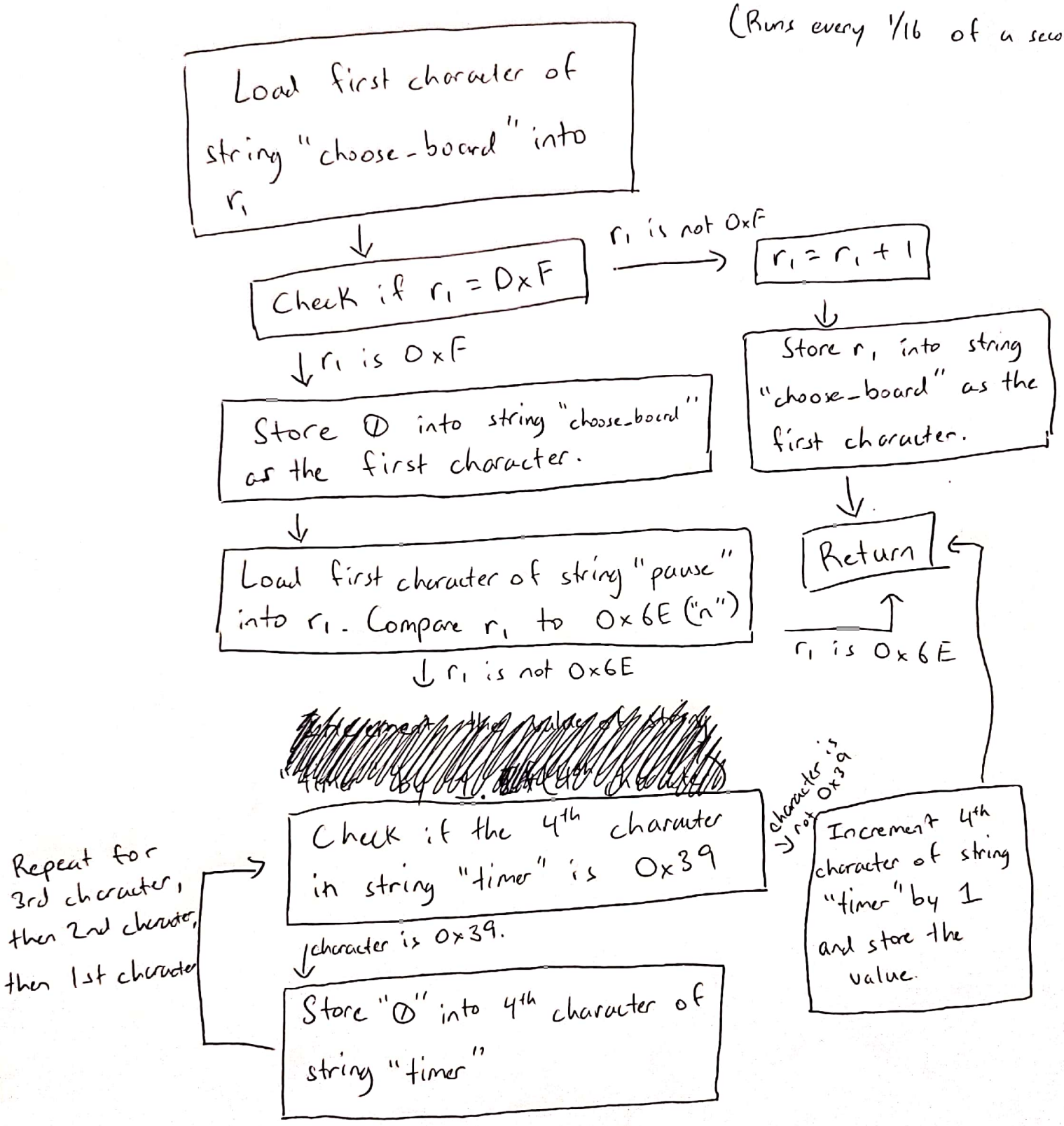
4.2 Switch_Handler

Switch_Handler



4.3 Timer_Handler

Timer Handler.
(Runs every 1/16 of a second).



5 Appendix

5.1 ASCII Key

The key of which the ASCII characters are converted into colors is listed below.

5.1.1 Dots

- Red Circle - 1
- Green Circle - 2
- Yellow Circle - 3
- Blue Circle - 4
- Magenta Circle - 5
- Cyan Circle - 6
- White Circle - 7

5.1.2 Vertical Pipes

- Red Pipe (Vertical) - ?
- Green Pipe (Vertical) - @
- Yellow Pipe (Vertical) - A
- Blue Pipe (Vertical) - B
- Magenta Pipe (Vertical) - C
- Cyan Pipe (Vertical)- D
- White Pipe (Vertical)- E

5.1.3 Horizontal Pipes

- Red Pipe (Horizontal) - F
- Green Pipe (Horizontal) - G
- Yellow Pipe (Horizontal) - H
- Blue Pipe (Horizontal) - I
- Magenta Pipe (Horizontal) - J
- Cyan Pipe (Horizontal)- K
- White Pipe (Horizontal)- L

5.1.4 Corners

- Red Corner - M
- Green Corner - N
- Yellow Corner - O
- Blue Corner - P
- Magenta Corner - Q
- Cyan Corner - R
- White Corner - S