Flow Free ARM

Michael Mu
Aryan Pandey

May 2, 2021

Contents

1 Division of Labor

1.1 Michael Mu

- Designed the main subroutine
- Half of UART handler (Cursor movement)
- Switch handler
- Pause menu
- Timer handler
- Counting colors
- output_colors_completed
- mem_copy
- randomizer
- reset_timer
- uart_init
- read_string
- read_character
- num_digits
- int2str
- read_from_push_btn
- interrupt_init
- timer_init

1.2 Aryan Pandey

- Designed the boards.
- Designed the lookup table.
- Half of UART handler (Drawing lines)
- output_board
- output_RGB
- illuminate_RGB_led

- get_position
- del_lines
- output_string
- output_character
- str2int
- gpio_init

Debugged, collaborated, and reviewed all of the code together and separately. The entire project was done while working at the same time, so there was constant input and feedback between both members of the team.

2 Program Overview

2.1 How to Use Program

How to operate the program.

1. Load and run the code in Code Composer Studio.
2. Set up Putty to communicate to the board using the serial connection type at a speed of 115,200.
3. A starting prompt will print out telling you to press the 'f' key to begin the game. After pressing, a random board is displayed and the game begins.
4. The game screen will print out the number of connected color lines, the time elapsed, and the 9×9 game board. There is an impassible border of 'X's along the edges of the game board, making the effective play range a 7×7 game board. There will be 14 circles on the game board, evenly split among 7 colors. The cursor begins in the center of the board.
5. The game controls are as follows:
    (a) Use the "WASD" keys to move the cursor up, left, down, and right, respectively.
    (b) Press the spacebar on a circle to begin drawing a line. This will also light up the LED on the board to the same color as the circle that the cursor is currently on.
    (c) Press the spacebar again to stop drawing. This will also turn off the LED on the board.
    (d) Press the switch key to pause the game.
    (e) Press the switch key again to unpause the game.

(f) In the pause menu, use the 'W' and 'S' key to move the cursor up and down to select a menu option.

(g) In the pause menu, press the spacebar to select whichever option your cursor is currently on.

6. The game rules are as follows:

   (a) Draw a connection between each pair of similarly colored circles in the game board.

   (b) To draw a line between each pair, you must press spacebar on a starting circle to start drawing, then press spacebar on the correspondingly colored circle to stop drawing. If you do not press spacebar on a circle to end the drawing, you will overrun the circle, which is not considered a completed correct line, though it will draw.

   (c) Lines cannot overlap one another. If you try to draw over a line, the older line will be erased to allow the new line to be drawn.

   (d) The same colored line cannot draw over itself.

   (e) Whenever a new line begins drawing, any same-colored lines will be erased.

   (f) When all seven pairs of similarly colored circles are connected, the board is finished and the game is won.

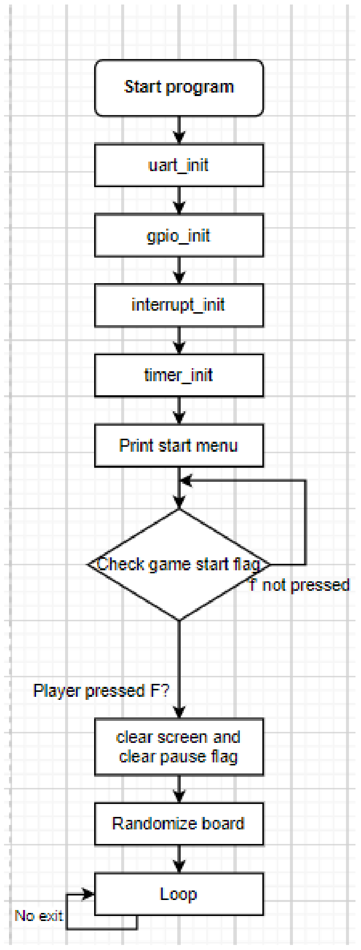
7. Once a game finishes, the timer will stop to report the time taken to beat the board. To begin a new game, press the switch button to open the pause menu. From the pause menu, you can restart the game with the same board, or randomly select a new board.

2.2 Program Summary

This program is the final lab of the semester and is an implementation of the popular game Flow. Using WASD, the spacebar, and the pushbutton on the physical board, a user can play the game by connecting the similarly colored circles displayed on the screen. When all of the circles are connected and the board is filled, the player wins. The game is viewable to the user through outputting the game board using PUTTY. The program also features a pause menu that can be used to resume, restart, or start an entirely new board. There's a timer as well as a completed colors counter that keeps track of the player's progress.

This program is a culmination of all the work and knowledge acquired throughout the semester. The game implements UART, GPIO, timers, subroutines, control flow, string manipulations, memory access, interrupts and handlers, and other techniques we've learned and implemented in previous labs. We also took advantage of the fact that many of the same techniques would be used, so we reused a library of prebuilt subroutines from previous labs.

2.3 Flowchart



3 Subroutine Descriptions

3.1 UART0_Handler

This subroutine interrupts the process whenever the user inputs a character into PuTTY. The UART handler operates in several different ways depending on the mode, which changes how it should interpret input from the keyboard. The UART handler is always guaranteed to clear the interrupt and read the input character. After this, it checks different flags to see if the game has started and if the game is paused. The operations performed will change depending on these modes.

3.1.1 Game Start

When the program first loads, there is a starting page with the instructions "Press the 'f' key to start:" The implication here is that no other input should start the game. Therefore, there is a flag to check when the game is started, and if the game has not started, none of the other interrupts can accept input. Timer handler will still run because we need to have some time elapse for our randomizer, but no input is accepted. In UART0_Handler, if the game flag shows the game has not started, the handler will only check to see if the input was the 'f' key. Otherwise, the handler will do nothing by skipping to the end of the handler. Once the 'f' key is pressed, the game started flag will be flipped to true and game initialization and normal game operation will begin as a result.

3.1.2 Unpaused

When the game is being played, the user can input W,A,S,D to move up, left, down and right respectively. The subroutine checks the character that was input, and jumps to the proper label accordingly. If an invalid input is given, the subroutine exits without performing anything.

If the spacebar is pressed,

- the character at the current cursor position is checked. If it is an O, its color is checked.
  - If the current color stored in memory does not match this color, it is stored as the new color in memory. We then check line completion conditions.
  - If the current direction of the player is null, the number of lines is not increased (player pressed spacebar twice.)
  - If the previous O position is the same as the current position, the number of lines is not increased (player connected line to same O.)
  - If these conditions are unsatisfied, the number of colors completed can be increased by 1, since the player has completed a line. The

new color is output through the RGB LED on board, and the board is printed again if required.

  - Upon reprinting the board, the cursor is first hidden. The number of lines completed, timer and the board is then printed, followed by the cursor being placed at the correct new position. The cursor is then revealed again using ANSI escape sequences.

- If the player presses spacebar on a space, the LED is turned off and the drawing "mode" is turned off.

If the WASD keys are pressed,

- The future position after movement is checked. If moving would make the players go out of bounds, OR if the player tries to move onto an O of a different color, OR if the player tries to move onto a line or cross of the SAME color being drawn, the subroutine exits without performing anything. The checks are performed using the lookup table values (see output_board.)
- The character at the next position is checked. If the player is not currently drawing (current_color=0), the cursor position in memory is modified, and the direction is updated, and the subroutine exits.
- If the character at the new position is an **O of the current color being drawn**, OR if the character at the new position is a **space**, a line or cross of the current color is drawn **if an O wouldn't be overwritten**, according to previous direction. output_board prints a new board to reflect changes.
- If the character at the new position is not a space, the color of that character is sent to del_lines. This deletes all lines of that color if the player is currently drawing.
- If cursor position was changed and the new position is not an O, it is replaced by a horizontal or vertical line.
- The cursor is first hidden, and the timer, colors completed and the board are outputted to the terminal if changes occurred.
- The new cursor position is output and then revealed again through an ANSI escape sequence.

At the end of the subroutine, the number of colors completed is checked. If all lines are completed, a flag is set and the timer is paused.

3.1.3 Paused

This section of the UART handler subroutine will run when the pause flag is set to true. In the pause menu, the user is presented with three options:

- Resume

- Restart Current Puzzle
- New Puzzle

The allowed inputs of in the pause menu that the UART0_handler tracks are the 'w' key, 's' key, and spacebar.

If the spacebar is pressed, the option that the cursor is currently on will be selected. The corresponding cursor location that is saved in memory will be checked to determine the next operation.

1. For option 1, or "resume", the program will simply skip past this part to the end of the subroutine where the pause flag is set to false.
2. For option 2, or "Restart Current Puzzle", the program will first load in from memory the selected puzzle board that had been last selected by the randomizer subroutine. Then, it will copy that board to the playing board in memory, effectively clearing the playing board to a starting position of the same board. The saved timer value in memory will then be reset to 0 to simulate a restart of the game. For both option 2 and option 3, the saved number of completed colored lines between endpoints and the timer then resets to 0 in memory since a new game is starting.
3. For option 3, or "New Puzzle", the program will first call the randomizer subroutine, which will select a new board pseudo-randomly based off of the current time elapsed since the start of the program. The randomizer subroutine will also take care of copying the newly selected game board into the playing game board. Afterwards, the saved timer value in memory will be reset to 0 to simulate a restart of the game. For both option 2 and option 3, the saved number of completed colored lines between endpoints and the timer then resets to 0 in memory since a new game is starting.

All of these options will result in the game resuming, whether its a new game or not. For this reason, at the end, the pause flag is set to false, or 0 regardless of the option selected.

If the 'w' key is pressed, the current option being selected is decremented unless the option is already the option 1, which is the top option and the lowest number possible. This happens by incrementing the corresponding value in memory as well as by getting the new cursor position based on the new option and outputting it. If the option is already option 1, then this part simply gets skipped since nothing changes.

If the 's' key is pressed, the current option being selected is incremented unless the option is already the option 3, which is the bottom option and the lowest number possible. This happens by incrementing the corresponding value in memory as well as by getting the new cursor position based on the new option and outputting it. If the option is already option 3, then this part simply gets skipped since nothing changes.

3.2 Switch_Handler

This subroutine interrupts the process whenever the SW1 switch on the board is pressed. First, the subroutine checks the flag that tracks whether the game started. If the game has not started, the switch handler skips to the end, not accepting or performing anything based on input. If the game has started, the subroutine continues to check the pause flag to see if the game is currently paused. If the game is currently unpaused, the program sets the pause flag to true (1), which will then make the game paused. The screen is then cleared by printing a new page, and the pause menu is printed out. The cursor is placed at option 1 by default. If the game is currently paused, the program sets the pause flag to false (0), which will then make the game unpaused. Nothing else is done because the timer handler will eventually overwrite the pause menu at the elapsed second interrupt.

Once the different operations are finished, the interrupt is cleared. This is done at the end in comparison to the other interrupts to prevent bounce issues, where the same button press is registered as several button presses and several interrupts are called consecutively. The added time before allowing a new interrupt from the pushbutton prevents the bounce issue.

3.3 Timer_Handler

This subroutine interrupts the process every 1 second (or 16 million ticks at 16MHz). It clears the interrupt before anything. Then, it checks two flags. The first flag is the pause flag. If the pause flag is true (1), then the subroutine skips to the end and doesn't perform any operations. The second flag is the completed game flag. If the flag is true (1), then all the lines are connected and the game is completed, so the timer handler again skips to the end and doesn't perform any operations. This stops the timer and allows the player to view their completion time once they beat the game.

If none of the flags are set to true, then the timer handler proceeds as normal. In preparation of updating the timer, the board is cleared by printing a new page. The subroutine loads the previous time saved into memory, which is saved as a string, converts the value to an integer using the str2int subroutine, and then increments that value by one. If the value happens to overflow past the timer size limit of 4 digits, then the timer resets from 9999 to 0. Before outputting, the cursor is hidden to prevent the cursor from jumping all over the board. Then, in order, the number of colors completed is calculated and outputted by calling the output_colors_completed subroutine, the time is converted to a string and outputted, and then the board is outputted by calling the output_board subroutine. Finally the cursor is placed by outputting the current position of the cursor that was saved into memory. The cursor is then shown again at the correct location.

3.4 output_colors_completed

This subroutine counts the number of lines that are confirmed to be connected. This is stored in the memory using a single byte that represents a bitmask, with each bit corresponding to a color being completed. The value would therefore be 01111111b if the board was completed.

Using a loop the subroutine counts the 1's that appear at bit 0, and right shifts each iteration. Once the entire byte is value at 0, the loop ends. The string that stores the number of 1's is updated to the new value and then outputted.

3.5 output_RGB

This subroutine handles the conversion between the RGB formats of ANSI and the LED on-board. This is done by simple switching bits 1 and 2 of the ANSI format. The subroutine utilizes the GPIO ports to output the correct color onto the board. If the player presses spacebar on an O, this function also saves the position of that O into memory if required.

3.6 get_position

This subroutine calculates and converts the 2-D cursor position saved in memory into a one-dimensional offset for easier memory manipulation. The output is in r0, and it represents the memory distance from the start of the board in the top left corner to the memory position where the cursor currently is. It uses the ANSI cursor position stored in memory and converts it into a single offset by calculating the memory of the previous rows and adding that to the memory of the previous columns. It takes the Time and Colors Completed row into account by subtracting 2 from the number of rows. This can be used in order to get the character at the current position, by a simple load instruction while using the returned value as a memory location offset.

3.7 del_lines

This subroutine takes an input in r0, which is the color of the lines to be deleted in its numerical representation, not the bitmask representation. This is accomplished by clearing the bit at the corresponding bitmask location for that color in memory. If the lines were previously connected, the subroutine reduces the count of colors completed. It then loops through the entire board, checking each character's value. If the color of the character matches with the input color, and if it is not an O, the value is replaced with a space. This loop continues until it reaches the end of the board.

3.8 mem_copy

This subroutine takes in an input r0 that is the memory address of the starting board that the program has selected to play in the randomizer subroutine. This subroutine runs in a loop that loads a byte from the source board provided by input r0 and then copies that same byte into the target playing board. Both memory locations increment for the next iteration. This subroutine only works for strings because it functions by looping through memory until it comes across the null terminator '\0', at which point the loop stops and the subroutine ends.

3.9 randomizer

This subroutine takes no inputs, but does utilize timer data. The first thing the subroutine does is load the time that has elapsed since the program began in the form of ticks. By getting the value of the first 4 bits, which is equivalent to modulus 16, a pseudorandom number from 0-15 can be acquired. Using this pseudorandom number a board numbered from 0 to 15 is chosen. That board is saved into memory as the selected board that will be played, and then that board is copied into the playing board using the mem_copy subroutine.

3.10 reset_timer

This subroutine takes no inputs and simply sets the time in memory back to 0. The timer in the board itself is not changed because we may want to generate a new random board using the board's time elapsed counter. This subroutine loads the point in memory where the time is being stored. Gets a new number valued at 0 and converts it to a string. Then the subroutine writes the new time that is valued at 0 to memory.

3.11 output_board

This subroutine outputs the board being played onto the PuTTY terminal. It utilizes the lookup table that was created for this project.

The table is as follows:
Example:
0x13/ 00010011
The above example converts to a yellow horizontal line, from the table below.

Symbol Table	
00	O
01	-
10	+
11	—

Color Table	
000	Black
001	Red
010	Green
011	Yellow
100	Blue
101	Magenta
110	Cyan
111	White

Special Characters	
0xFF	X
0x20	Space
10_{10}	Newline
13_{10}	Linefeed

The subroutine utilizes the above mapping, and loops through the board stored in memory. Using conditional checks, it outputs the correct ANSI escape sequences if it needs to output a color, and prints the corresponding characters onto the terminal. The loop exits upon reaching a null terminator.

3.12 uart_init

This subroutine initialized all of the values needed to begin serial communication through UART0. By setting the right values through the memory map I/O, the clock to UART0 was enabled, the clock to PortA was enabled, the UART configurations were set, the system clock was set, UART0 was enabled, and PA0 and PA1 were set as digital ports with alternate function for UART.

3.13 gpio_init

This subroutine initializes the GPIO Port F pins to allow the SW1 button to act as an input, and the on-board RGB LEDs to act as an output. This is done by enabling the clock, setting the GPIO Port F pin directions, enabling the pull-up resistor and setting pins 1 through 4 to digital.

3.14 timer_init

This subroutine initializes the built-in timer by enabling it's clock. It then disables the timer in order to set its configuration to 32-bit, enable periodic mode and set the correct timer interval load to 16 million. The subroutine returns after enabling the timer again.

3.15 interrupt_init

This subroutine does additional configurations that relate to the previous init subroutines to allow each peripheral to interrupt the processor. The receive interrupt mask bit in the UART interrupt mask register is set. The processor is configured in the Interrupt 0-31 Set Enable Register (EN0) so that the 30th (GPIOF), 19th (TIMER0A), and 5th (UART0) are all set to 1 to allow them to interrupt the processor. We set the GPIO to be edge-sensitive, turn off both edges, and only enable rising edge to trigger an interrupt. Finally, we set the receive interrupt mask bit in the GPIOM register for GPIOF pin 4.

3.16 read_character

This subroutine reads in a single character from the memory mapped UART data register. It accomplishes this by waiting until the receiving bit flag in the UART flag register is set to be not empty, signifying that there is data to be read, then copying the value from the data register.

3.17 output_character

This subroutine takes a character's ASCII value as an input and outputs it onto the PuTTY terminal. It uses a loop that checks the UART Flag Register to see whether the transmitter is full. This is achieved by loading in the value at the Flag Register and isolating the TxFF bit. If the transmitter is empty, the loop exits and the character is written onto the UART Data Register, which is then transmitted onto the terminal.

3.18 read_string

This subroutine constantly reads in characters using the read_character subroutine from user input in a loop until the user presses the enter key. Each character is stored into memory, and after the input stops, a null terminator is appended.

3.19 output_string

This subroutine takes an input in r0 as a pointer to the string that is to be output. The subroutine calls output_character in a loop until it reads a null character (0x00). Each character can be seen in the PuTTY terminal.

3.20 read_from_push_btn

This subroutine checks the GPIO data register's value for the 4th pin (which corresponds to the SW1 button on-board) to see if the button is being held down or not. The function returns 1 if it is being held down, and 0 if not.

3.21 illuminate_RGB_LED

This subroutine takes an input in r0, with a size of 3 bits. Bit 0 corresponds to red, bit 1 corresponds to blue, and bit 2 corresponds to green. Since this is already the way that the pins are laid out in memory, the subroutine sets the corresponding GPIO data register values to turn on the RGB LED on board according to the input color.

3.22 num_digits

This subroutine takes an integer stored within a register and calculates the number of digits within it. It goes through a loop where the number is divided by 10 in each iteration, and exits when 0 is reached, with a counter for each iteration.

3.23 int2str

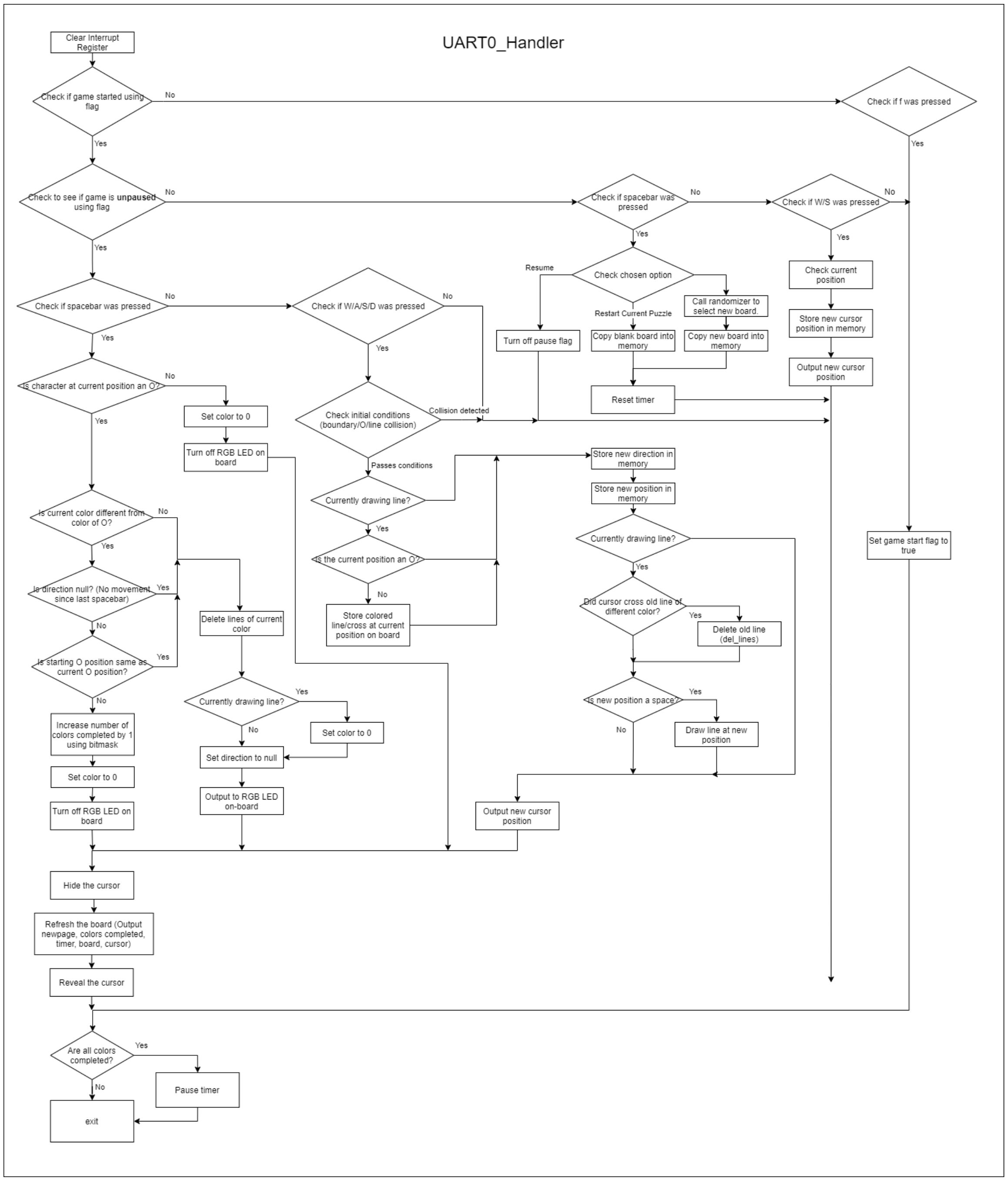
This takes an integer stored within a register, along with the number of digits and a pointer to where this integer will be stored as a string. This is achieved by isolating each digit and adding the ASCII offset of 0x30, which is then written onto the correct memory location, along with a null character at the end. If the value was originally negative, a negative sign is appended to the string at the end.

3.24 str2int

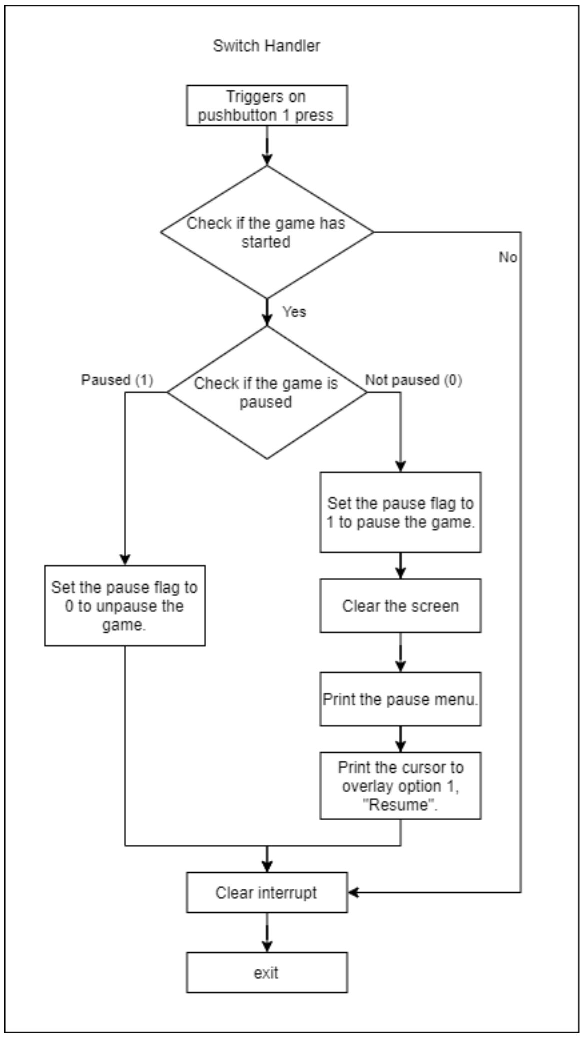
This subroutine was reused from Lab3 and modified to be generalized as a library call. The subroutine now takes the pointer to a string in r0, and reads each character through a loop until it reaches null. The ASCII offset of 0x30 is subtracted from the character's value to give us the digit within the number. If there was a negative sign in the beginning, the value is multiplied by -1 at the end.

4 Additional Flowcharts

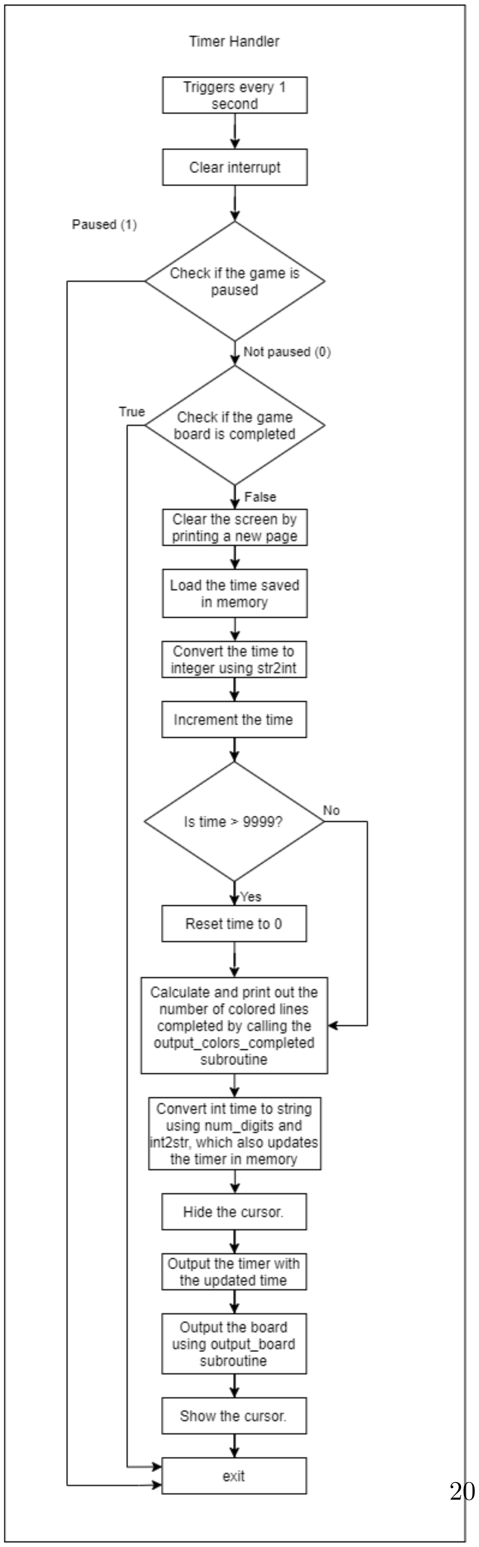
4.1 Subroutine UART0_Handler


UART0_Handler
Clear Interrupt Register
Check if game started using flag
No
Check if f was pressed
Yes
Yes
Check to see if game is **unpaused** using flag
No
Check if spacebar was pressed
No
Check if W/S was pressed
No
Yes
Yes
Check current position
Store new cursor position in memory
Output new cursor position
Check chosen option
Resume
Restart Current Puzzle
Call randomizer to select new board.
Turn off pause flag
Copy blank board into memory
Copy new board into memory
Reset timer
Yes
Check if spacebar was pressed
No
Check if W/A/S/D was pressed
No
Yes
Yes
Is character at current position an O?
No
Set color to 0
Turn off RGB LED on board
Check initial conditions (boundary/O/line collision)
Collision detected
Passes conditions
Yes
Currently drawing line?
Yes
Is the current position an O?
No
Store colored line/cross at current position on board
Store new direction in memory
Store new position in memory
Currently drawing line?
Yes
Did cursor cross old line of different color?
Yes
Delete old line (del_lines)
Is new position a space?
Yes
No
Draw line at new position
Output new cursor position
Is current color different from color of O?
No
Yes
Is direction null? (No movement since last spacebar)
Yes
No
Is starting O position same as current O position?
Yes
No
Delete lines of current color
Currently drawing line?
Yes
No
Set color to 0
Set direction to null
Output to RGB LED on-board
Increase number of colors completed by 1 using bitmask
Set color to 0
Turn off RGB LED on board
Set game start flag to true
Hide the cursor
Refresh the board (Output newpage, colors completed, timer, board, cursor)
Reveal the cursor
Are all colors completed?
Yes
Pause timer
No
exit

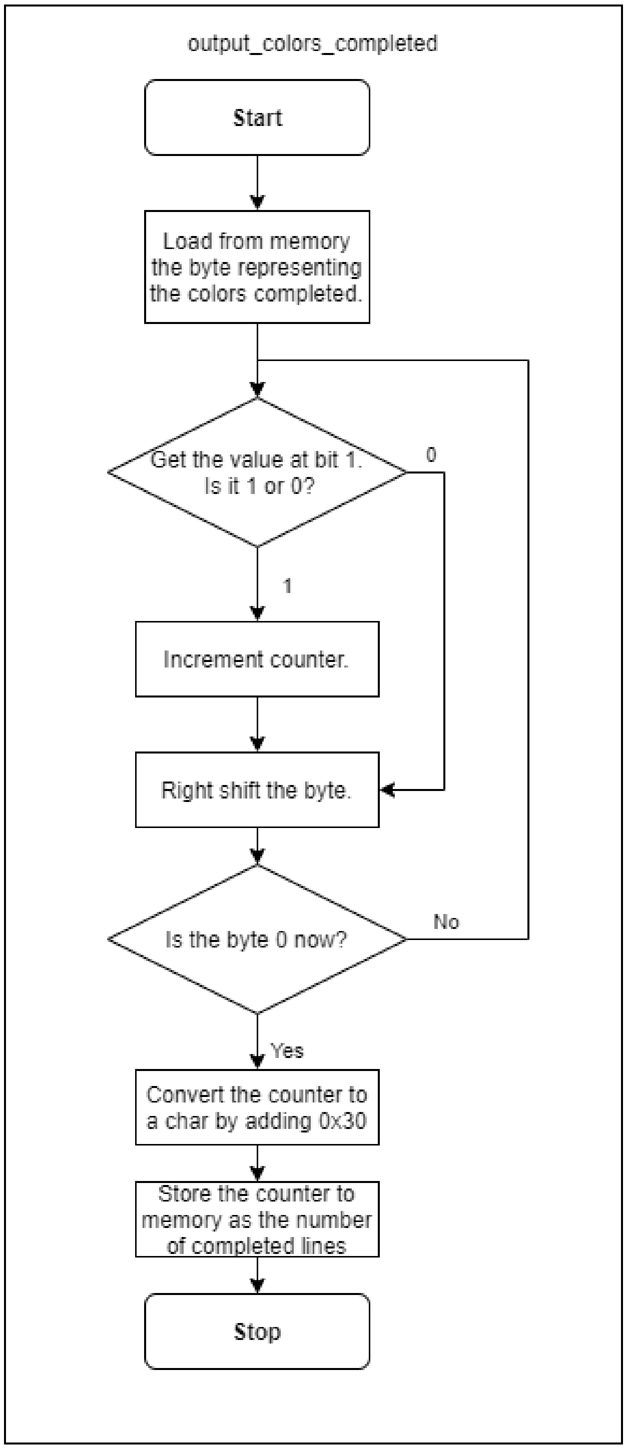
4.2 Subroutine Switch_Handler


Switch Handler
Triggers on pushbutton 1 press
Check if the game has started
No
Yes
Check if the game is paused
Paused (1)
Not paused (0)
Set the pause flag to 1 to pause the game.
Clear the screen
Print the pause menu.
Print the cursor to overlay option 1, "Resume".
Set the pause flag to 0 to unpause the game.
Clear interrupt
exit

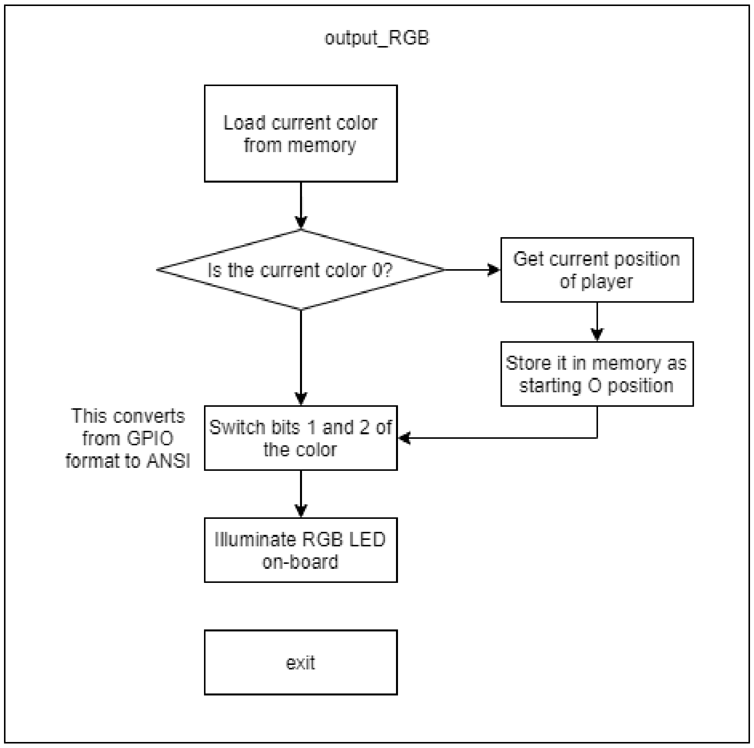
4.3 Subroutine Timer_Handler


Timer Handler
Triggers every 1 second
Clear interrupt
Paused (1)
Check if the game is paused
Not paused (0)
True
Check if the game board is completed
False
Clear the screen by printing a new page
Load the time saved in memory
Convert the time to integer using str2int
Increment the time
Is time > 9999?
No
Yes
Reset time to 0
Calculate and print out the number of colored lines completed by calling the output_colors_completed subroutine
Convert int time to string using num_digits and int2str, which also updates the timer in memory
Hide the cursor.
Output the timer with the updated time
Output the board using output_board subroutine
Show the cursor.
exit

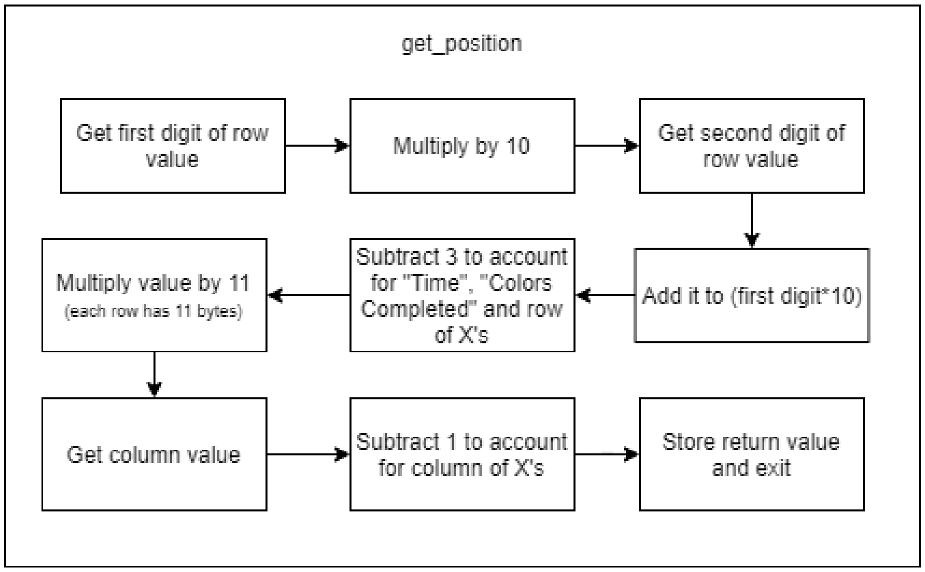
4.4 Subroutine output_colors_completed


output_colors_completed
Start
Load from memory the byte representing the colors completed.
Get the value at bit 1. Is it 1 or 0?
0
1
Increment counter.
Right shift the byte.
Is the byte 0 now?
No
Yes
Convert the counter to a char by adding 0x30
Store the counter to memory as the number of completed lines
Stop

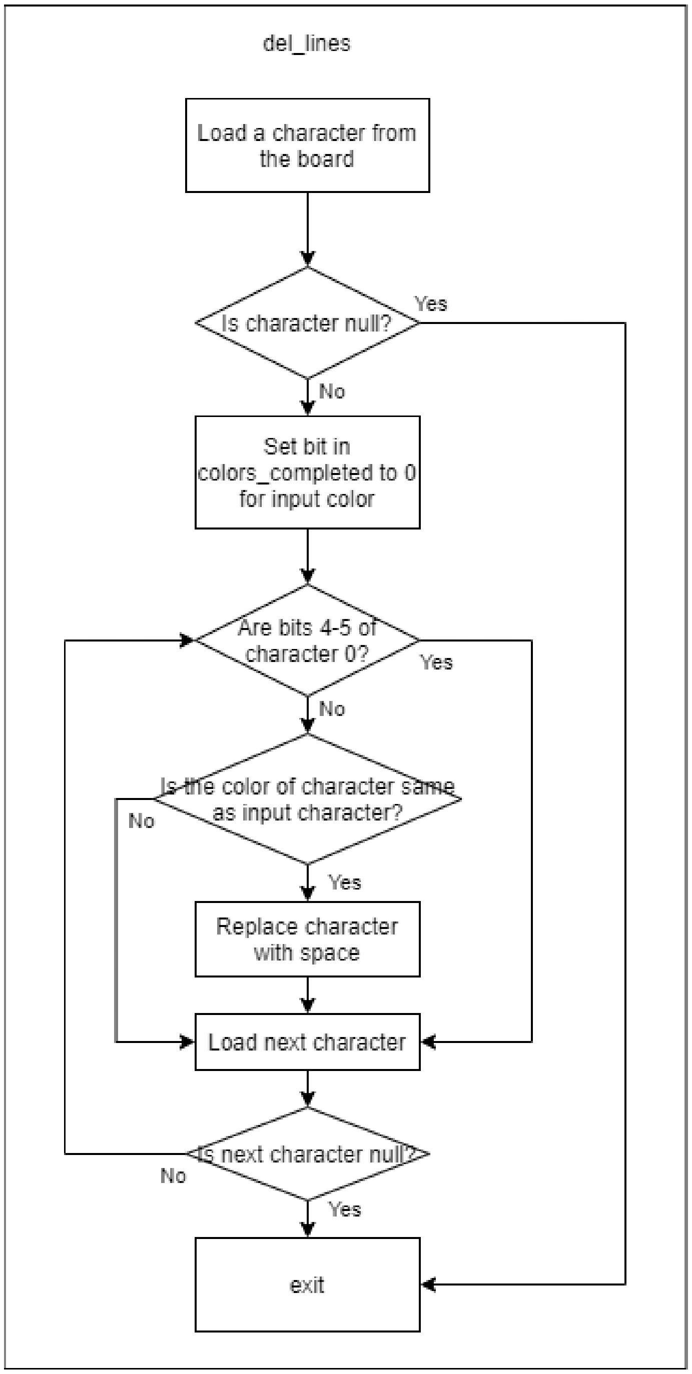
4.5 Subroutine output_RGB


output_RGB
Load current color from memory
Is the current color 0?
Get current position of player
Store it in memory as starting O position
This converts from GPIO format to ANSI
Switch bits 1 and 2 of the color
Illuminate RGB LED on-board
exit

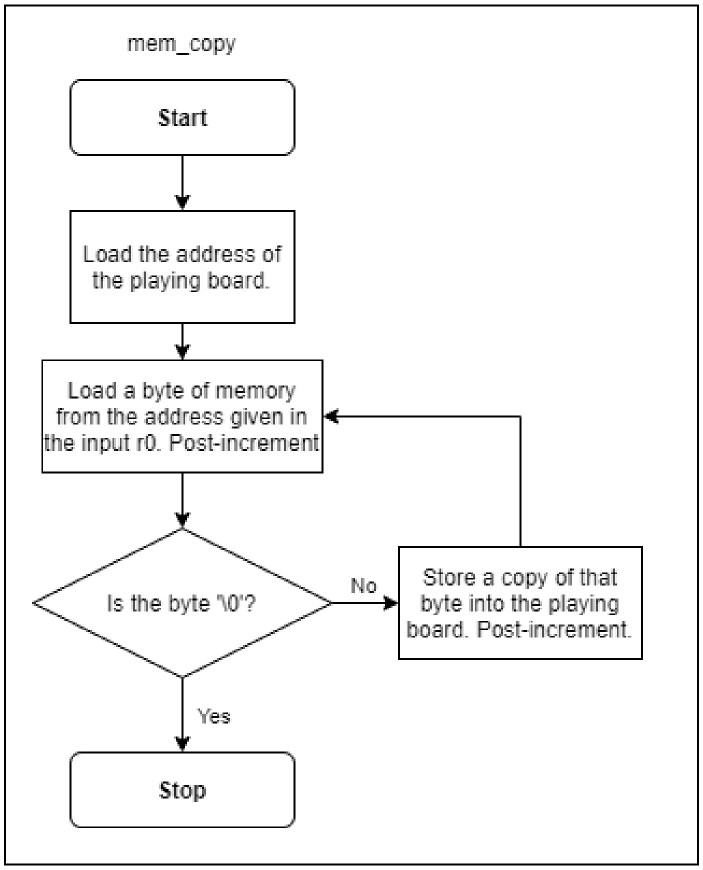
4.6 Subroutine get_position


get_position
Get first digit of row value
Multiply by 10
Get second digit of row value
Add it to (first digit*10)
Subtract 3 to account for "Time", "Colors Completed" and row of X's
Multiply value by 11 (each row has 11 bytes)
Get column value
Subtract 1 to account for column of X's
Store return value and exit

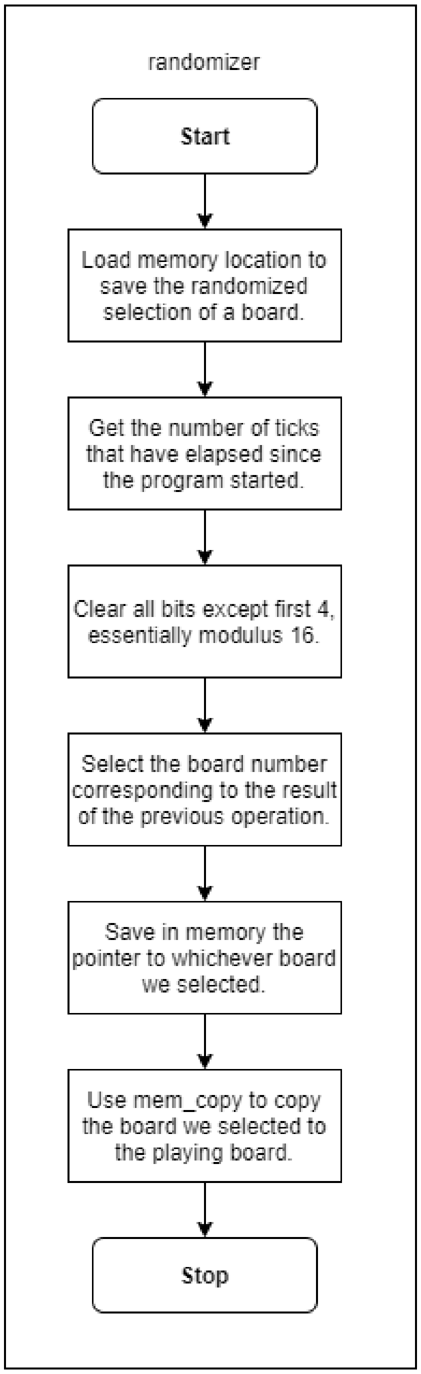
4.7 Subroutine del_lines


del_lines
Load a character from the board
Is character null?
Yes
No
Set bit in colors_completed to 0 for input color
Are bits 4-5 of character 0?
Yes
No
Is the color of character same as input character?
No
Yes
Replace character with space
Load next character
Is next character null?
No
Yes
exit

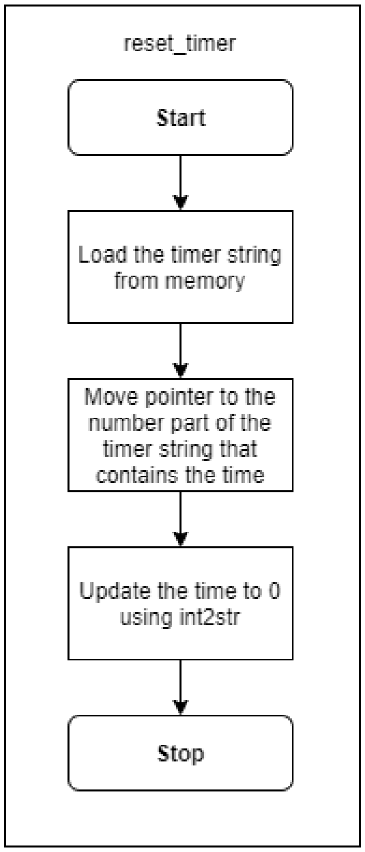
4.8 Subroutine mem_copy


mem_copy
Start
Load the address of the playing board.
Load a byte of memory from the address given in the input r0. Post-increment
Is the byte '\0'?
No
Store a copy of that byte into the playing board. Post-increment.
Yes
Stop

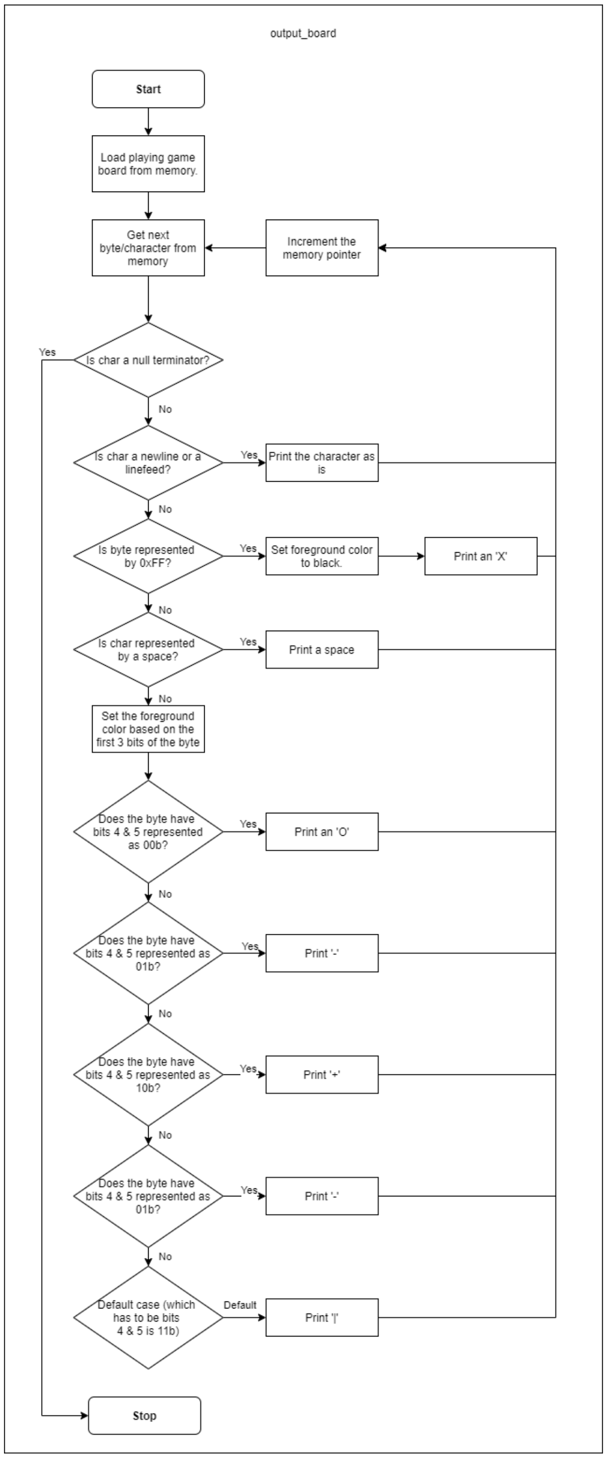
4.9 Subroutine randomizer


randomizer
Start
Load memory location to save the randomized selection of a board.
Get the number of ticks that have elapsed since the program started.
Clear all bits except first 4, essentially modulus 16.
Select the board number corresponding to the result of the previous operation.
Save in memory the pointer to whichever board we selected.
Use mem_copy to copy the board we selected to the playing board.
Stop

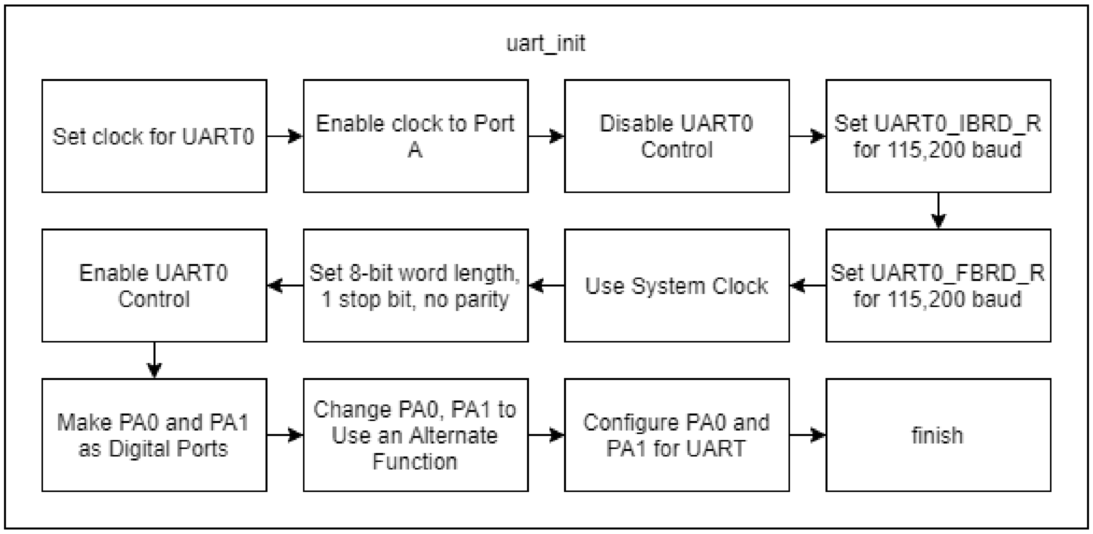
4.10 Subroutine reset_timer


reset_timer
Start
Load the timer string from memory
Move pointer to the number part of the timer string that contains the time
Update the time to 0 using int2str
Stop

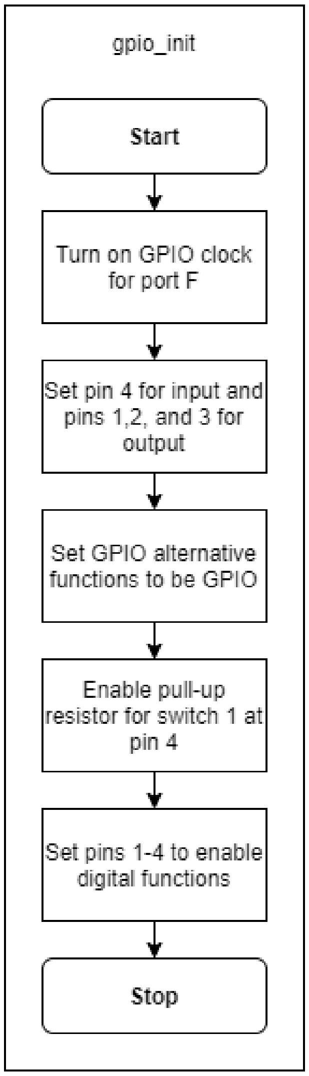
4.11 Subroutine output_board


output_board
Start
Load playing game board from memory.
Get next byte/character from memory
Increment the memory pointer
Is char a null terminator?
Yes
No
Is char a newline or a linefeed?
Yes
Print the character as is
No
Is byte represented by 0xFF?
Yes
Set foreground color to black.
Print an 'X'
No
Is char represented by a space?
Yes
Print a space
No
Set the foreground color based on the first 3 bits of the byte
Does the byte have bits 4 & 5 represented as 00b?
Yes
Print an 'O'
No
Does the byte have bits 4 & 5 represented as 01b?
Yes
Print '-'
No
Does the byte have bits 4 & 5 represented as 10b?
Yes
Print '+'
No
Does the byte have bits 4 & 5 represented as 01b?
Yes
Print '-'
No
Default case (which has to be bits 4 & 5 is 11b)
Default
Print '|'
Stop

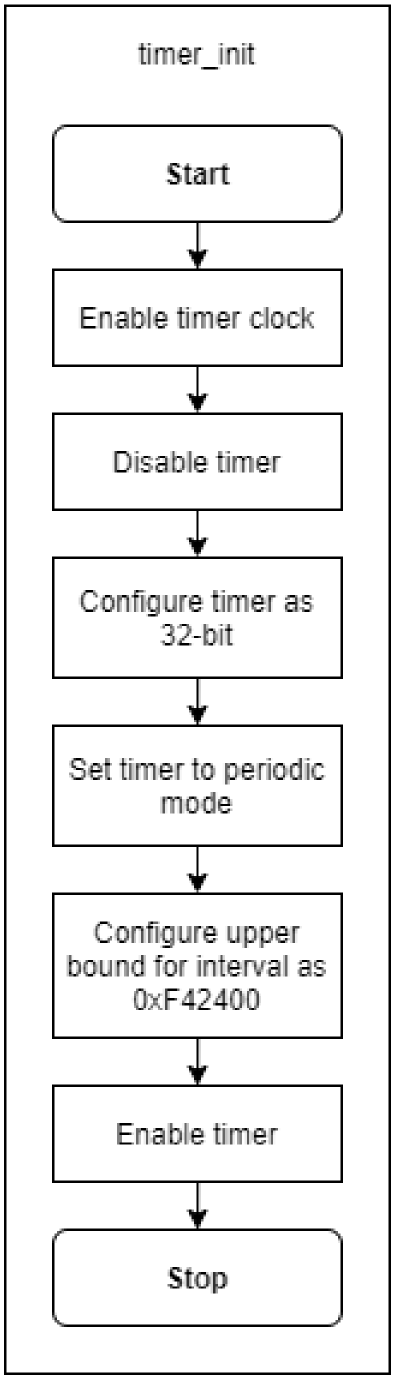
4.12 Subroutine uart_init


uart_init
Set clock for UART0
Enable clock to Port A
Disable UART0 Control
Set UART0_IBRD_R for 115,200 baud
Set UART0_FBRD_R for 115,200 baud
Use System Clock
Set 8-bit word length, 1 stop bit, no parity
Enable UART0 Control
Make PA0 and PA1 as Digital Ports
Change PA0, PA1 to Use an Alternate Function
Configure PA0 and PA1 for UART
finish

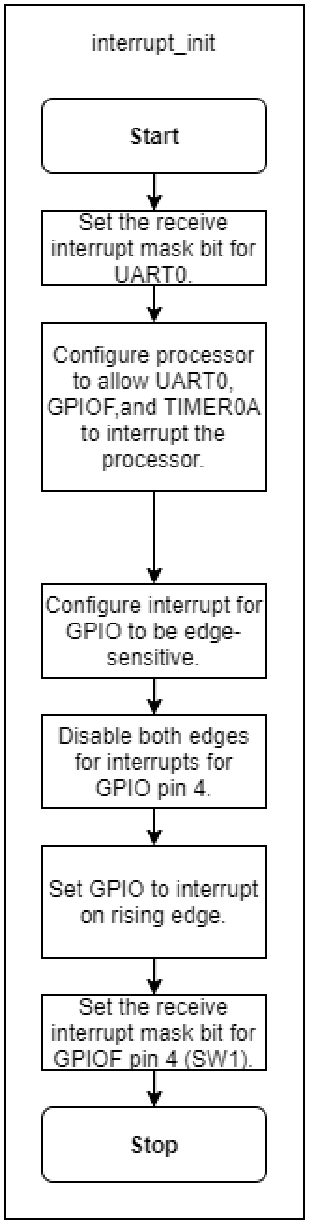
4.13 Subroutine gpio_init


gpio_init
Start
Turn on GPIO clock for port F
Set pin 4 for input and pins 1,2, and 3 for output
Set GPIO alternative functions to be GPIO
Enable pull-up resistor for switch 1 at pin 4
Set pins 1-4 to enable digital functions
Stop

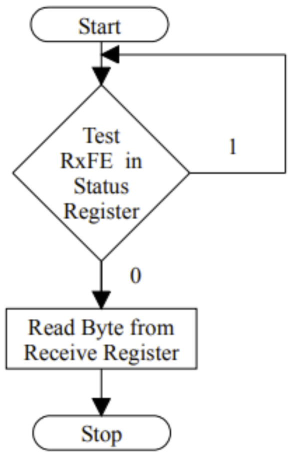
4.14 Subroutine timer_init


timer_init
Start
Enable timer clock
Disable timer
Configure timer as 32-bit
Set timer to periodic mode
Configure upper bound for interval as 0xF42400
Enable timer
Stop

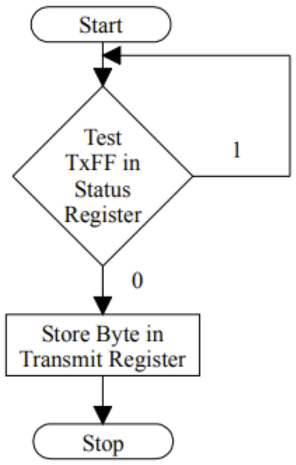
4.15 Subroutine interrupt_init


interrupt_init
Start
Set the receive interrupt mask bit for UART0.
Configure processor to allow UART0, GPIOF,and TIMER0A to interrupt the processor.
Configure interrupt for GPIO to be edge-sensitive.
Disable both edges for interrupts for GPIO pin 4.
Set GPIO to interrupt on rising edge.
Set the receive interrupt mask bit for GPIOF pin 4 (SW1).
Stop

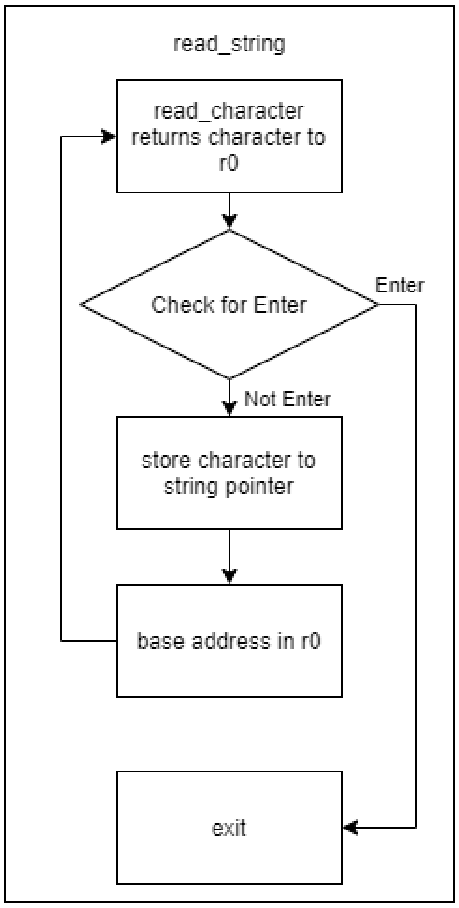
4.16 Subroutine read_character



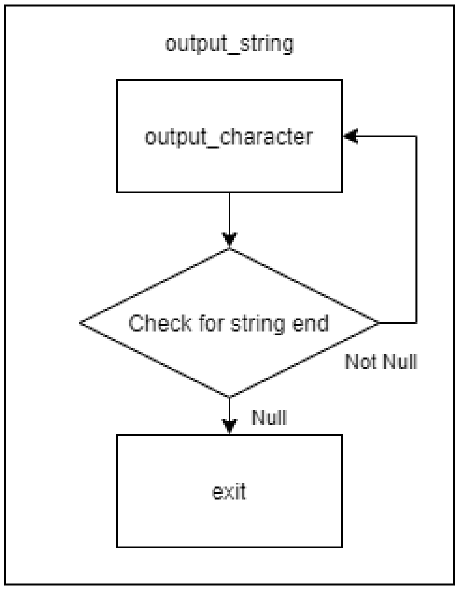
4.17 Subroutine output_character



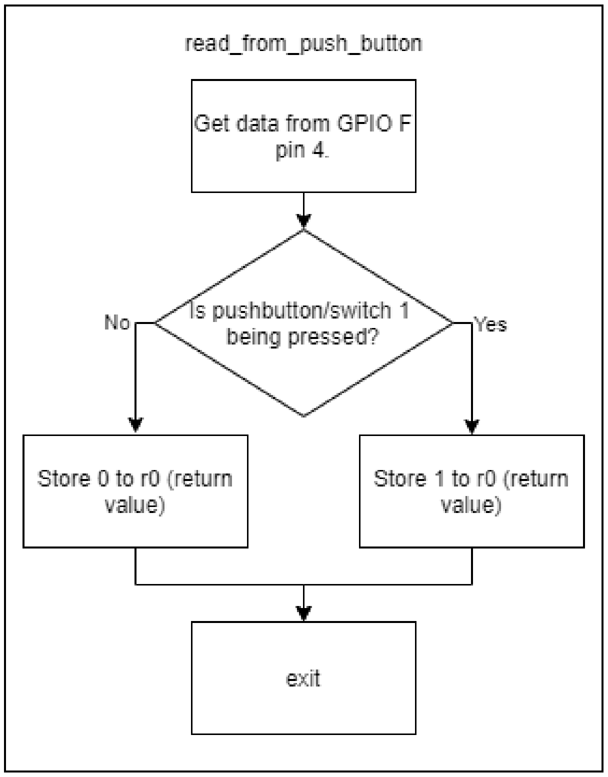
4.18 Subroutine read_string


read_string
read_character returns character to r0
Check for Enter
Enter
Not Enter
store character to string pointer
base address in r0
exit

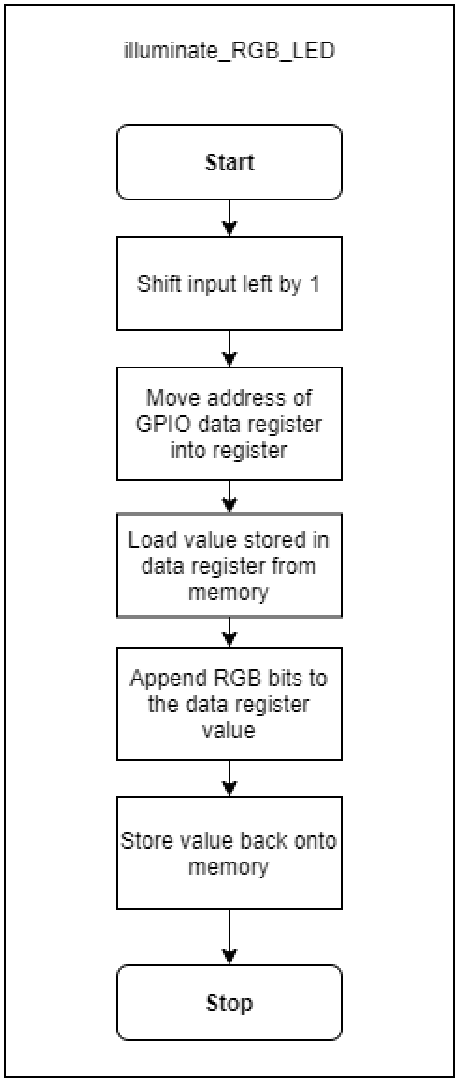
4.19 Subroutine output_string



4.20 Subroutine read_from_push_btn

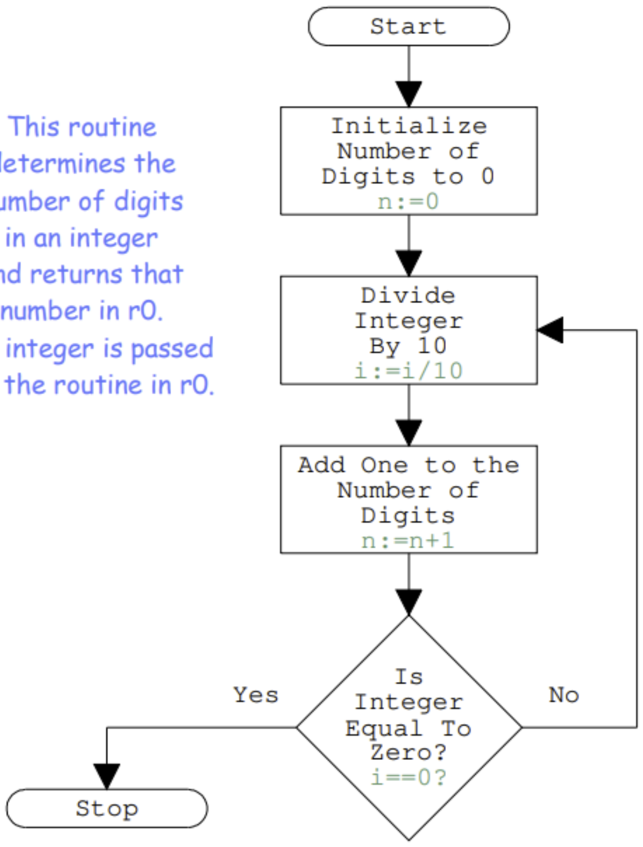

read_from_push_button
Get data from GPIO F pin 4.
Is pushbutton/switch 1 being pressed?
No
Yes
Store 0 to r0 (return value)
Store 1 to r0 (return value)
exit

4.21 Subroutine illuminate_RGB_LED

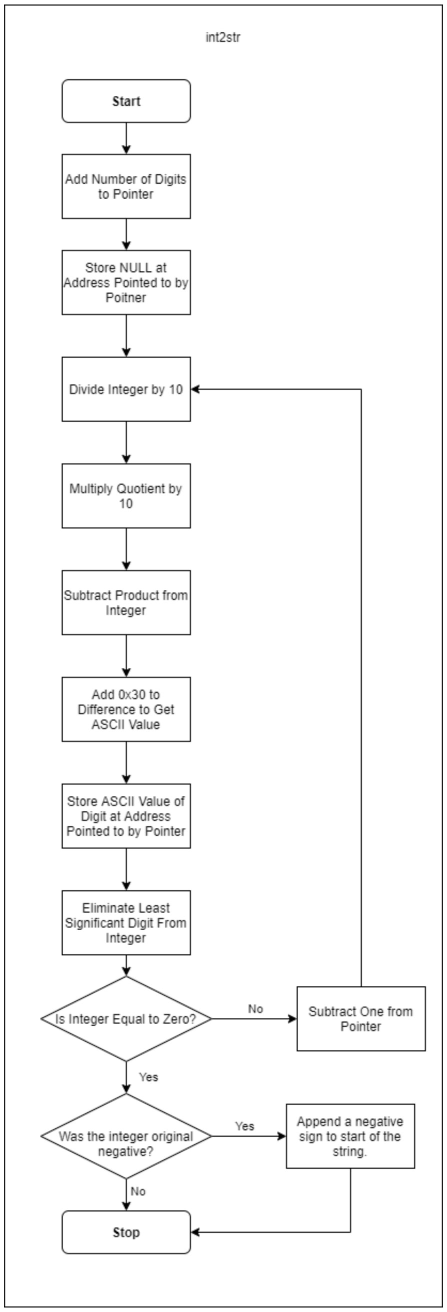

illuminate_RGB_LED
Start
Shift input left by 1
Move address of GPIO data register into register
Load value stored in data register from memory
Append RGB bits to the data register value
Store value back onto memory
Stop

4.22 Subroutine num_digits

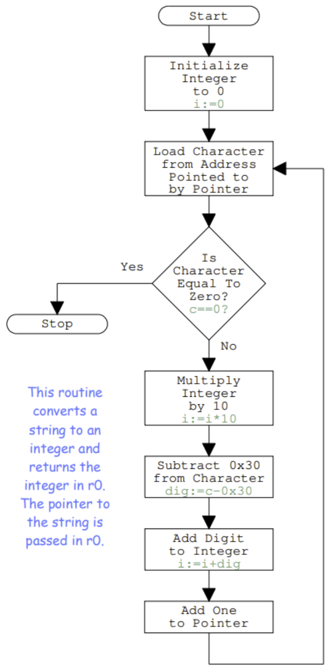
This routine determines the number of digits in an integer and returns that number in r0. The integer is passed into the routine in r0.


Start
Initialize Number of Digits to 0
n:=0
Divide Integer By 10
i:=i/10
Add One to the Number of Digits
n:=n+1
Is Integer Equal To Zero?
i==0?
Yes
No
Stop

4.23 Subroutine int2str


int2str
Start
Add Number of Digits to Pointer
Store NULL at Address Pointed to by Poitner
Divide Integer by 10
Multiply Quotient by 10
Subtract Product from Integer
Add 0x30 to Difference to Get ASCII Value
Store ASCII Value of Digit at Address Pointed to by Pointer
Eliminate Least Significant Digit From Integer
Is Integer Equal to Zero?
No
Subtract One from Pointer
Yes
Was the integer original negative?
Yes
Append a negative sign to start of the string.
No
Stop

4.24 Subroutine str2int


Start
Initialize Integer to 0
i:=0
Load Character from Address Pointed to by Pointer
Is Character Equal To Zero?
c==0?
Yes
Stop
No
Multiply Integer by 10
i:=i*10
Subtract 0x30 from Character
dig:=c-0x30
Add Digit to Integer
i:=i+dig
Add One to Pointer
This routine converts a string to an integer and returns the integer in r0. The pointer to the string is passed in r0.