Software PDR

**Creation of Gamebox portable gaming system**

**using STM32F407**

Jake Bonte, Joseph Krambeer, William Jensen

University of St. Thomas

ENGR 331-01

Professor Nepal

Spring 2016

Abstract

Our game system is designed to be able to play games on a hand held, battery powered device. The device needs to be able to take in user input through buttons on the device and display the game on an LCD. The algorithms detailed in this document will be used to both interface with hardware components of the device and also implement the rules of our games. To streamline the process of interfacing with the hardware of the system, we have broken down our algorithms for hardware into: Power Level Detection, Input Signal Handling, Graphics Output, and Audio Output. The most difficult algorithm in this system will be implementing the game logic in a manner that works in unison with the hardware algorithms and allows for intuitive play of the game. To tackle this challenge we will make the scale of the game very minimal while still making good use of the hardware aspects of our device and providing entertaining gameplay. By using effective algorithms we will be able to accomplish this task of making an entertaining gaming system.

Table of Contents

**ABSTRACT**….……………………….……………………….………………………………………………….…………………………**2**

**ALGORITHMS**……………………………..……….………………………………………….……………………………….………..**4**

Power Level Detection….…………………………………………….………………………………………………………..**4**

Power Saving Mode Enable….…………………………….……….…………………………………………..…………...**6**

Input Signal Handling…………………………………………..………………………………..……………….….………...**8**

Graphics Output......…………………..…………………….……………………………………………….…..….……….**10**

Audio Output...…………..…………………..….………………..................………….…….………….…..………….**15**

Game Logic…….………………………………….………….…………………………………………………….…………….**17**

Game Description………………………………………………………………………………………………………**17**

Rules Handling………………….……………………………………………………………………………………….**18**

**REFERENCES**………………………………………………….…………….…………………………………………….…………….**21**

Algorithms:

Power Level Detection:

To detect a lower power in the system, the Power Controller (PWR) on the STM board will be used to occasionally check the operating voltage that the board is running off of. Depending on how much voltage the board has it will send signals on two output pins to enable or disable transistors connected to the power status LED. For sufficient levels the LED should output a green color light and for insufficient levels the LED should output a red color light.

To turn the circuits on a logic 1 should be sent to the transistors that control the circuits and to turn them off they should receive a logic 0.

Steps:

1. Enable PWR’s clock
   1. Write PLS bits to set voltage threshold the power voltage detector will detect (will detect drops below 2.7V of onboard current)
   2. Write 1 to PVDE bit to enable voltage detector.
2. Enable port B’s clock
   1. Set PB6 and PB7 as outputs
   2. PB6 will be used to control the green LED and PB7 will be used to control the red LED
3. Enable timer 4’s clock
   1. Set its pre-scalar to 15999
   2. Set auto-reload register to 999
   3. Enable timer 4’s interrupt (TIM4\_IRQHandler)
4. In timer 4’s interrupt
   1. Check PVDO bit in PWR\_CSR register (Check this)
   2. If PWR\_CSR is set to 0 the voltage is about 2.7V, so output a 1 on the pin controlling the transistor for the green LED, enabling the circuit, and output a 0 on the pin controlling the transistor for the red LED, disabling the circuit.
   3. If PWR\_CSR is set to 1 the voltage is at or lower than 2.7V, so output a 0 on the pin controlling the transistor for the green LED, disabling the circuit, and output a 1 on the pin controlling the transistor for the red LED, enabling the circuit.

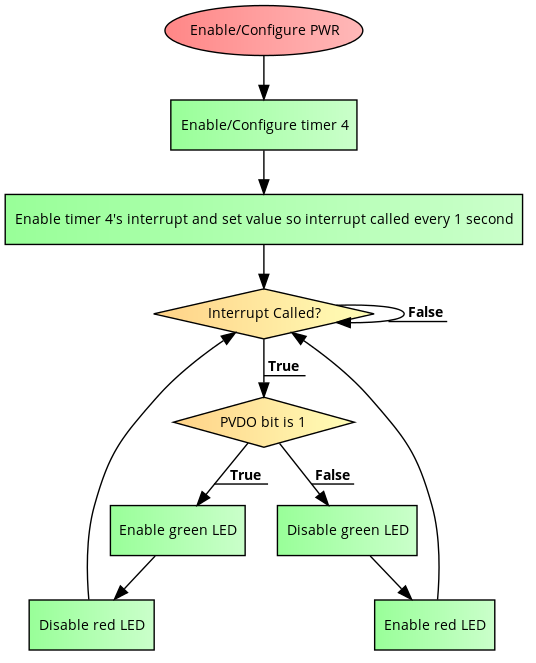


Diagram for power-level detection

Power Saving Mode Enable:

To enable/disable the backlight on the LCD, the user will be able to press a button to toggle the whether the LCD is enabled or disabled. Pressing this button will toggle the state of an output pin that is connected to the logic controlling the backlight of the display, and depending on the state the signal will either enable or disable the backlight on the display. The button presses will be read in on an input pin and will trigger and interrupt to change the output of the pin that controls the backlight circuit.

Steps:

1. Enable clock on port E
   1. Set PE6 as output and PE5 as input (also described in input signal handling)
   2. Set PE5 to be pull down
2. Enable Timer 6 to delay set amount of times (used again in input signal handling)
   1. Set pre-scalar to 15999
   2. Make method that takes in value for ARR that sets ARR to value-1 to delay how many milliseconds the function was called with
3. Set output state on PE6 so that backlight circuit is enabled
4. Enable external interrupt for PE5 (EXTI0\_IRQHandler)
5. When button, the external interrupt handler for PE5 is called
   1. Clear pending interrupts on this interrupt handler
   2. In interrupt handler, wait 1ms with Tim6 delay to de-bounce button
   3. Toggle of PE6 state by XOR’ing it with 1, toggling the state of the backlight
6. Continue execution of other code, with interrupt being called when button controlling power saving mode is pressed.

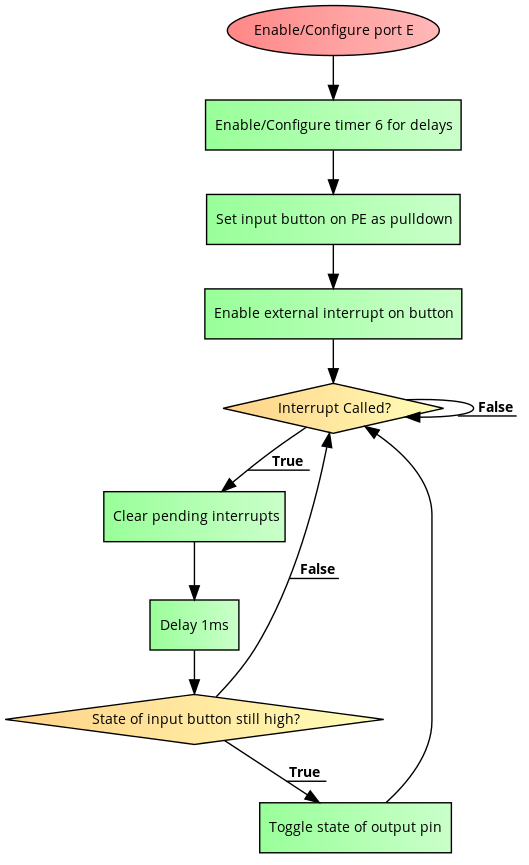


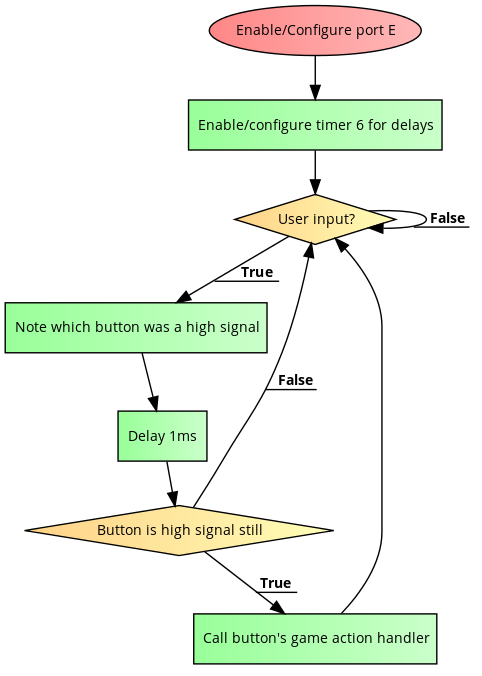
Diagram for power-saving mode

Input Signal Handling:

To enable the user to interact with the game system, the input signal processing must allow the user to control the game in an intuitive way without any anomalies like double presses or presses not registering. Waits on user input, de-bounces it, and then handles input with corresponding game action if it still there after de-bounce.

Steps:

1. Enable port E’s clock
   1. Set pins PE0-PE5 as inputs
   2. PE0 will be used for the top button on the D-pad
   3. PE1 will be used for the left button on the D-pad
   4. PE2 will be used for the bottom button on the D-pad
   5. PE3 will be used for the right button on the D-pad
   6. PE4 will be used for game actions like breaking blocks (detailed in game rules)
   7. PE5 will be used for toggling on/off power-saving mode
   8. Set these pins to pull down mode
2. Enable Timer 6 to delay set amount of time
   1. Set pre-scalar to 15999
   2. Make method that takes in value for ARR that sets ARR to (value-1) to delay how many milliseconds the function was called with
3. Wait until input is detected in port’s Input Data Register
   1. Wait 1ms with timer 6 delay to de-bounce signal
   2. See if signal is still a high, and if it is, handle signal for that button with corresponding action in updating game state
4. Go back to step 3 and wait on user input



Input signal handling diagram

Graphics Output:

To interface with our TFT-LCD display we are going to be using an SDD1963 controller. This controller interfaces with the LCD and helps makes writing commands to the display simpler, and thus better graphics can be developed to be used on the display (these graphics are shown below in the game description). This controller for the LCD requires a 20 pin interface to write commands and data to the screen, so multiple GPIO ports will be utilized for display functions. To make interfacing easier, methods will be made to send data, commands, and initialize the LCD will be made.

Initializing ports for LCD interfacing:

1. Enable port E’s clock
   1. Set pins PE7-PE14 as output pins for 8 data lines to SDD1963
2. Enable port D’s clock
   1. Set pins PD9-PD13 as output pins
   2. PD9 will be used as RS line (register select)
   3. PD10 will be used as WR (write signal)
   4. PD11 will be used as RD (read signal)
   5. PD12 will be used as CS (chip select signal)
   6. PD13 will be used as REST (reset signal)
3. Set PE6 as output to use as backlight enable switch (described in power saving mode)
4. Enable Timer 6’s clock
   1. Set pre-scalar to 15999
   2. Make method that takes in value for ARR that sets ARR to value-1 to delay how many milliseconds the function was called with

Steps to send data to LCD controller:

1. Write data on port E’s 8 data lines (PE7 is LSB, PE14 is MSB)
2. Output 0 on PD9 (Low RS)
3. Output 0 on PD12 (Low CS)
4. Output 0 on PD10 (Low WR)
5. Wait 1 clock cycle (No instruction)
6. Output 1 on PD10 (High WR)

Steps to send commands to LCD:

1. Write command on port E’s 8 data lines (PE7 is LSB, PE14 is MSB)
2. Output 1 on PD9 (High RS)
3. Output 0 on PD12 (Low CS)
4. Output 0 on PD10 (Low WR)
5. Wait 1 clock cycle (No instruction)
6. Output 1 on PD10 (High WR)

Steps for initialize LCD controller:

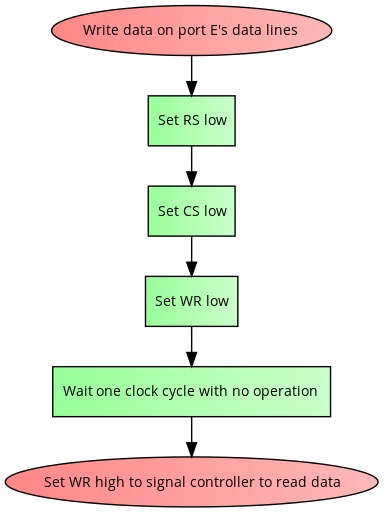
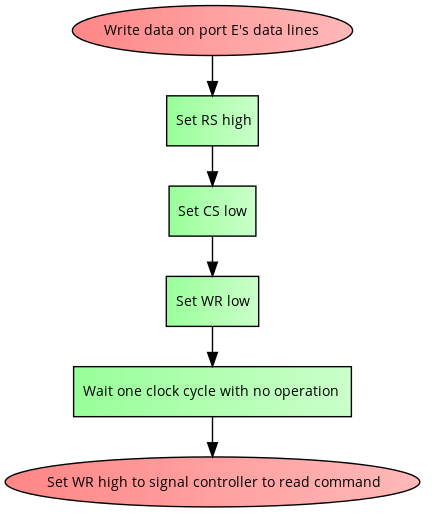
1. Output 0 on PD13, delay 2ms, then output 1 on PD13 to reset controller
2. Write software reset command to controller (0x01)
3. Write start PLL command to controller (0xE0)
4. Write data to configure PLL clock and SYS clock if necessary (no changes from defaults for our system)
5. Delay, write start command to set PLL settings (0xE0)
6. Set TFT to 18 bit mode (0xB0)
7. Configure TFT mode, data format (RGB 6/6/6 with command 0x3a then command 0x60, horizontal/vertical size, column/page addresses (column 0-319, page 0-239) with data sent over. (Ex. send 0x13f to set horizontal size 320 pixels, 0xef to set page size as 240 pixels).
8. Write command for normal mode (0x13)
9. Write command for display on (0x29)
10. Start writing other commands/data to display

Steps for writing pixel colors:

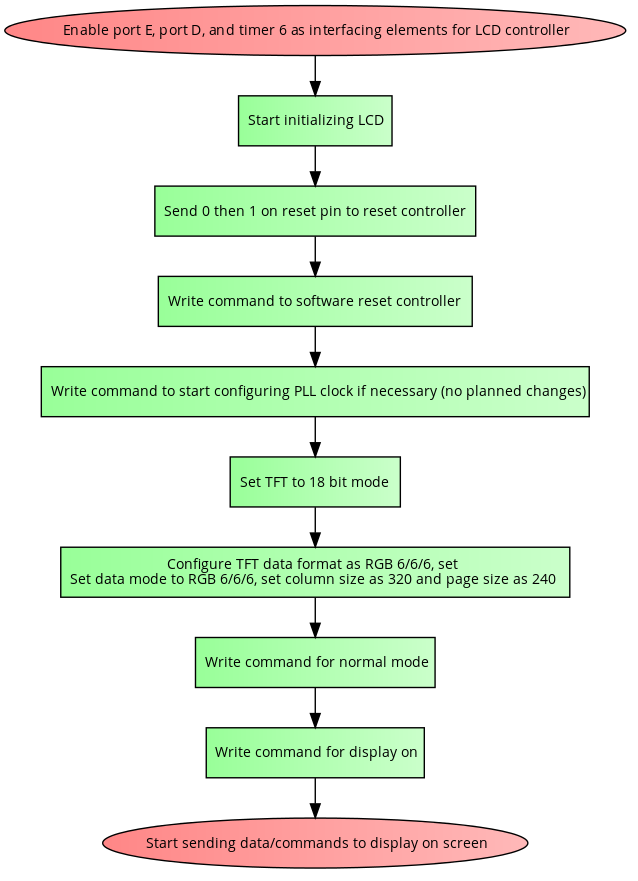
1. Set column address (0x2A) and send data for pixel's column position
2. Set page address (0x2B) and send data for pixel's page position
3. Write command to start memory write (0x2C)
4. Write data for RGB value that pixel should have

Steps for writing game status to screen:

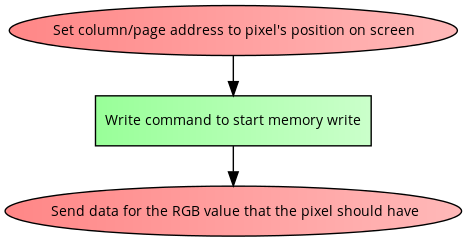
1. Loop i=0 to 319
2. Inside i loop, loop j=0 to 239
3. For every i,j pair find its corresponding color by checking the tile it lies within and then call method to write pixel with the RGB value it should have



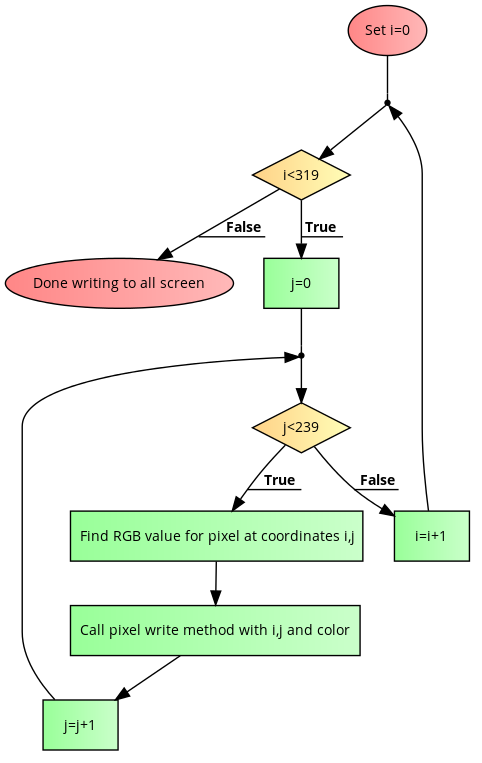
Write Command Method Write Data Method



Startup sequence for LCD



Write color to specific pixel



Writing to all pixels on screen

Audio Output:

In our system, one song will continually be played looping continually. To minimize the impact that playing audio will have on our system, the audio algorithm must not take time so it doesn’t impact game play in any harmful way. The audio output will be output through the Digital to Analog Converter on the STM board, using the DAC’s output to create a sine waves with varying frequencies to make tones on the speaker. To output a sine wave, the audio will reference an array with a sine wave cut in 128 parts and use the voltage level at that part as a reference to what the DAC should output. To change the frequency the DAC outputs the wave at, a timer with varying values it counts up to will be used trigger changing values the DAC’s output.

Steps:

1. Enable DAC’s clock
   1. Set DAC to trigger enable outputs
   2. Set DAC to output on software trigger
   3. Disable output buffer
   4. Enable DAC channel 1
2. Enable timer 7’s clock
   1. Set pre-scalar to 0
   2. Set auto reload register to first value in delays
   3. Enable timer interrupt
   4. Enable one pulse mode on timer
   5. Enable counting
3. Enable port A’s clock
   1. Set PA4 as analog output for DAC channel 1
4. Enable interrupt for timer 7 (TIM7\_IRQHandler)
5. Create array that holds a 128 part sine wave
6. In timer 7’s interrupt
   1. Keep track of 3 counts to keep track what part of sine wave is being output, index of the current delay in delays array, and how many times a full sine wave has been output at the current frequency.
   2. Clear status flag in timer’s SR register
   3. Re-enable counting on timer (may be changed later to conditionally re-enable counting to enable periods of silence on the speaker)
   4. Put the next sine part in the DAC channel 1 right aligned register
   5. Write 1 to SWTRIGR register in DAC to output value in register you just wrote to
   6. If last part of sine wave was just output, increment count to how many times a full sine wave has been output at this frequency and if it gets above the value it should count to, set it to 0
   7. Check against how many full sine waves should be output, and if there have been that many output, update the timer’s ARR with the new value for a different frequency and update index for which delay is being used
   8. Update index of for which part of sine wave is being output next and if it is >127 set it to 0

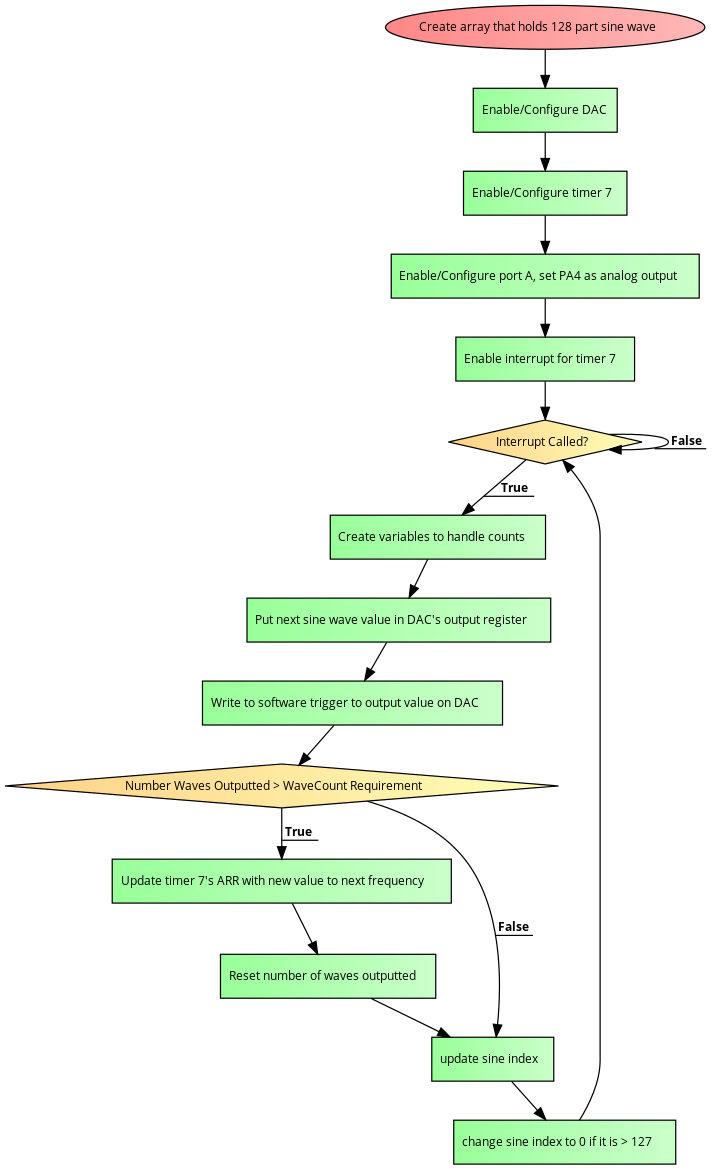
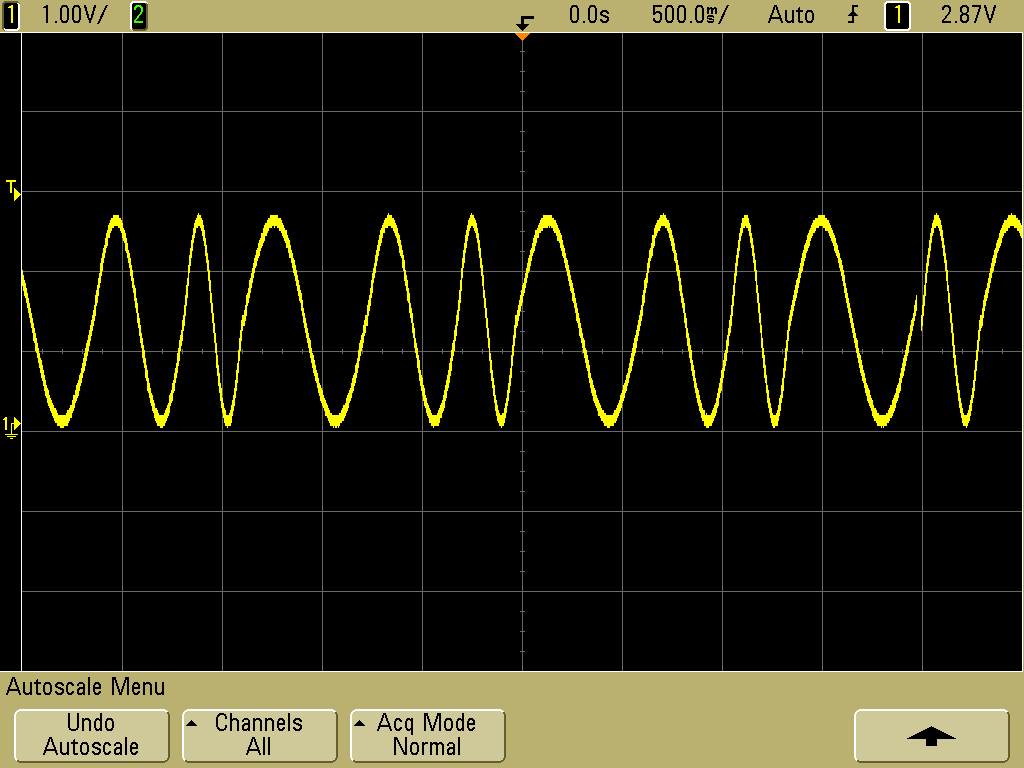


Diagram for audio wave creation



Example varying frequency sine wave being output on PA4 (DAC channel 1)

Game Logic:

Game Description:

The game that we will first implement on our system consists of a tile based system were the goal of the player is to move once, and only once, across all solid black tiles on the screen putting green ‘X’ marks on them. The player will control their movement by the inputs give on the D-pad of the device, and inputs on the buttons will also impact the game. Whenever the player moves, if they are allowed to move into that tile, the tile they moved onto will be marked with a green ‘X’ if it was a black tile before, and the player icon will moved to the appropriate position. The player will not be able to move off the screen, and any movement that would move them off screen will be ignored. In each level, in addition to the solid black tiles, there will be magma tiles, ice tiles, breakable tiles, direction tiles, and immovable tiles. Magma tiles are tiles that the player can move into but once they enter they ‘die’ and must restart the level from the start again. Ice tiles are tiles the player can move into but will continue moving the direction they entered the tile from until they hit a non-ice tile. Breakable tiles are tiles the player cannot move into, but by pressing a primary button when next to a breakable tile the player can ‘break’ them and turn them into a normal black tile. Directional tiles are moveable tiles that when the player enters them they must go the direction the arrow points. Finally immovable tiles are tiles that the player cannot move into and will be used as obstacles to make moving to every tile more difficult for the player. If a player moves into a tile with a green ‘X’ in it they must restart the level from the start position again.



Example screen from the game (Level isn’t solvable)

In this example of how the game will work, the red tiles are magma, the blue tiles are ice, the orange tiles are breakable tiles, the yellow arrows are direction tiles, and the grey tiles are the immovable tiles. The player in this example would start on the tile outline in light-green dots.

\*Note: rules many be slightly modified in final project but the object of the game will always be to move along a ‘maze’ and visit every tile exactly once.

Rules Handling:

To make sure the game runs as described previously, the system will need to keep track of game state, and enforce the rules of the game to make sure it plays as expected. To do so the game will keep track of a 20 by 15 grid (tiles correspond to 16 pixel by 16 pixel chunks of the screen). The game will also keep track of the players x and y coordinate to know where to display the player's icon and to be able to apply rules to the player's movements.

Steps for overall game logic:

1. Place character in their starting x/y coordinate on the map (differs map to map)
2. Wait on user input
3. When user input is encountered:
   1. If left D-pad button was pressed call its handler
   2. Else if right D-pad button was pressed call its handler
   3. Else if top D-pad button was pressed call its handler
   4. Else if bottom D-pad button was pressed call its handler
   5. Else if primary button 1 was pressed call break blocks method
4. Check if tile player is now on is magma or a previously visited tile
   1. If they are, clear all visited flags and put player back on starting coordinated
5. Check if all tiles are visited, update the level if there is another one
6. Refresh screen to show new player position
7. Go back to step 2

Steps for D-pad input handlers:

1. Depending on button press check tile at certain coordinate
   1. If left, if player x is 0 do nothing, else check tile at player x-1,y
   2. If right, if player x is 19 do nothing, else check tile at player x+1,y
   3. If up, if player y is 0 do nothing, else check tile at player x,y-1
   4. If down, if player y is 14 do nothing, else check tile at player x,y+1
2. If tile at checked position is moveable place the character there, update the player's x/y and update the tile's flag for visited if necessary
3. If the tile player is now on is ice, check tiles in the direction they input until an ice tile before an immovable tile, a moveable tile, or a tile on the map edge is encountered and place the player updating their x/y accordingly
4. If the player moved into a directional tile, treat it as a another button input in the direction the arrow is pointing and go back the 1st step
5. Refresh the screen (write to all pixels on screen, detailed in Graphics section)

Steps for break blocks method:

1. If it exists, check the block at player x-1,y and if it is breakable update it in the map array with a solid black movable tile
2. If it exists, check the block at player x+1,y and if it is breakable update it in the map array with a solid black movable tile
3. If it exists, check the block at player x,y-1 and if it is breakable update it in the map array with a solid black movable tile
4. Check block at player x,y+1 and if it is breakable update it in the map array with a solid black movable tile
5. If a tile was changed, refresh the screen to display the change

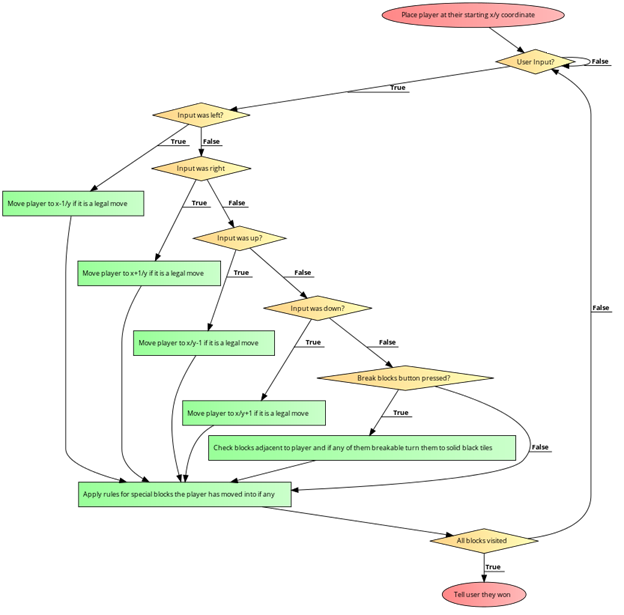


Diagram for basic game logic with user input

References:

Generating sine wave on STM board: <http://amarkham.com/?p=49>

Display controller datasheet: <http://www.newhavendisplay.com/app_notes/SSD1963.pdfs>

Display controller board datasheet: <http://www.newhavendisplay.com/specs/NHD-3.5-320240MF-20%20Test%20Board.pdf>

Example code for SDD1963 LCD controller:

<https://newhavendisplay.com/appnotes/excode/txt/TFT/TFT_35M.txt>

Example of video game source code we looked at (game is DOOM for Linux): <https://github.com/id-Software/DOOM>