

# CSEE4840 Embedded Systems

## Final Report: Tanks

Quinn Booth (qab2004)  
Ganesan Narayanan (grn2112)  
Ana Maria Rodriguez (amr2343)

May 11, 2023

### Table of Contents

<b>1 Introduction.....</b>	<b>2</b>
1.1 Game Overview + Rules.....	2
1.2 System Architecture.....	2
<b>2 Hardware.....</b>	<b>4</b>
2.1 Graphics.....	4
2.3 Audio.....	6
2.4 Memory.....	9
<b>3 Software.....</b>	<b>11</b>
3.1 User Input.....	11
3.1.1 Overview.....	11
3.1.2 Communication Protocol.....	12
3.2 Game Logic.....	13
3.2.1 Overview.....	13
3.2.2 Tank Movement.....	14
3.2.3 Collision Detection.....	14
3.2.4 Bullet Firing + Movement.....	14
3.2.5 End Game.....	14
3.3 Avalon Bus Interface.....	15
<b>4 Discussion.....</b>	<b>16</b>
4.1 Challenges.....	16
4.2 Lessons Learned + Advice.....	16
4.3 Who did what section?.....	17
<b>5 References.....</b>	<b>18</b>
<b>6 Code.....</b>	<b>19</b>

# 1 Introduction

## 1.1 Game Overview + Rules

Our game of Tanks is a 2-player tank maze game based on the original Tank arcade game developed in 1974 by a subsidiary of Atari. In our game, two players move tanks around in a maze viewed from above, while attempting to shoot the opposing player's tank. Players use game controllers to control their tank, moving with the arrow buttons and shooting bullets with the A button. Bullets cannot go through walls and when a bullet hits the other player's tank, it explodes and they gain 100 points. The first player to reach 500 points wins.

Upon startup, Player 1 must select the map to be played. We have designed three different maps for the players to select. Using the up/down arrows on the controller, they can select from the desired stage, with higher number stages being more complex in maze design. To start the game, Player 1 must press the A button. Once a player reaches 500 points and wins, the game is over and the players are taken back to the stage selection screen.

## 1.2 System Architecture

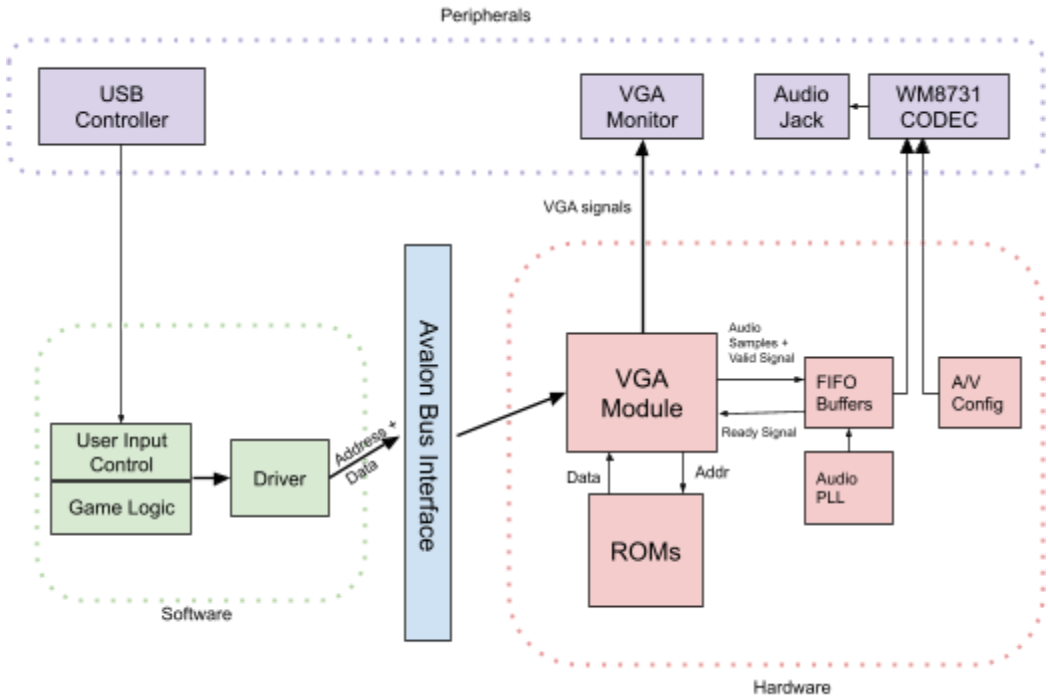


Figure 1.2: Block Diagram

Players control the movement of tanks using game controllers, interfacing with the software controlling the game logic by communicating through the USB protocol. The software then communicates to the FPGA hardware using a device driver, which then handles displaying the graphics for the game on the VGA monitor and decides when to play audio cues.

The software components involve the main game logic `hello.c` file, which handles the logic of tank movement, bullet shooting, and scoring. The `controller.c` file recognizes and initializes inputs from the USB controller controllers so that the game logic can be carried out, communicating through the USB protocol and `libusb` library. Finally, the `vga_ball.c` file device driver communicates to the `vga_module.sv` through the Avalon bus interface to update the graphics that will be displayed on the VGA monitor based on the game logic.

The hardware peripherals include the USB controllers, through which players input is passed, and the VGA monitor, which displays the output of the game itself. The hardware consists of on-chip memory ROMs on the FPGA in which all the necessary sprite data will be stored and the `vga_ball` module file that displays the requisite graphical information based on the screen location on the VGA display. The `vga_ball` module sends addresses to the ROMs which returns the requested output sprite data. It then communicates with the VGA monitor hardware peripheral to display the graphics. Additionally, we connect earbuds or a speaker to the WM8731 CODEC.

# 2 Hardware

## 2.1 Graphics

The main hardware algorithm is the logic to display the graphics. The sprites we used for our graphics are stored in on chip memory ROMs created and configured through the on-chip system memory IP blocks in Platform Designer.



Figure 2.1a: ROM configuration in Platform Designer

The .png images for our sprites (taken from the Battle City game) were converted into .mif memory initialization files to prepopulate the ROMs.

```
1  WIDTH = 8;  
2  DEPTH = 4096;  
3  ADDRESS_RADIX = DEC;  
4  DATA_RADIX = HEX;  
5  CONTENT BEGIN  
6  
7  0 : F;  
8  1 : F;  
9  2 : F;  
10 3 : F;  
11 4 : F;  
12 5 : F;  
13 6 : F;  
14 7 : F;  
15 8 : F;  
16 9 : F;
```

Figure 2.1b: Memory configuration file for sprite

To display the graphics, the addresses to be read from the respective ROMs are determined by the `ioctl` writes from the device driver and location on the display. The values stored in the ROMs specified by the `.mif` files are a hex value for each pixel in the image, with each pixel corresponding to an address in memory. This output hex value is used to look-up the values to pass to the VGA RGB signals and determine the color to display on the screen (for a total of 16 different colors) at that current location.

## 2.2 Graphics Display Architecture

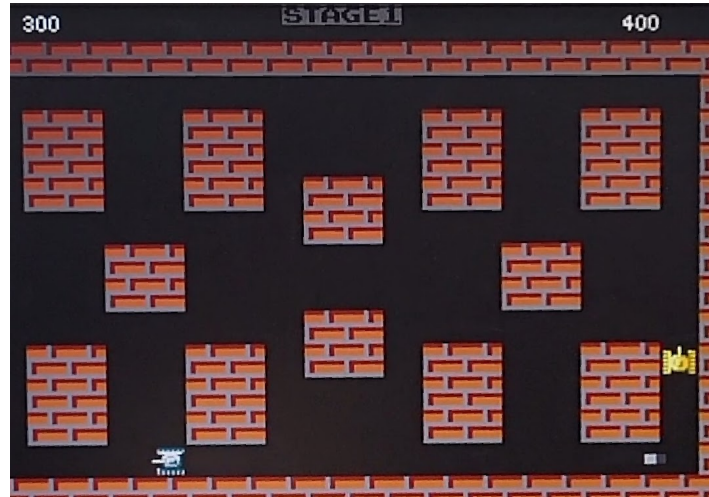


Figure 2.2a: Image of our game

The graphics architecture is shown below. The `vga_ball` module contains modules to determine the position on the screen and to set the RGB pixel values on the VGA display. Through the Avalon bus interface, 16 bit data is passed from the software using the device drivers to `vga_ball.sv` to indicate the when and where graphics should be displayed. Using this information and the `hcount` and `count` coordinate positions, the `vga_ball.sv` passes addresses to the instantiated on-chip memory ROMs, which return 8-bit output values that are used to determine the output `VGA_R`, `VGA_G`, `VGA_B` signals to the display. The ROMs are initialized with the memory initialization files that populate the memory contents with the requisite sprite data.

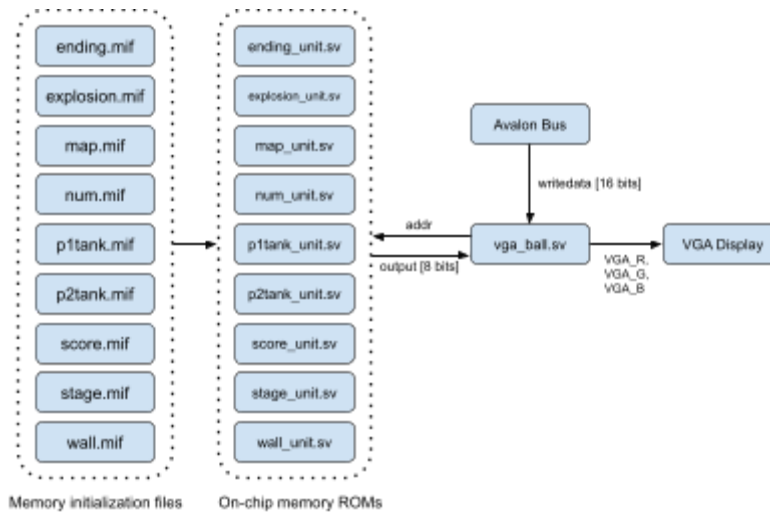


Figure 2.2b: Graphics architecture

## 2.3 Audio

To store the audio files in memory, we had to convert them to a specific formatting. After downloading the .mp3 files we wanted to use in our game, we converted them to .wav files in Audacity. We also swapped the files from stereo to mono (as we will only be playing one stream of audio), cropped them to a desirable length (to save memory), re-sampled the audio at 8 kHz, and converted them to signed 16-bit PCIM binary encoding. The .mp3s were then converted into .mif file format.

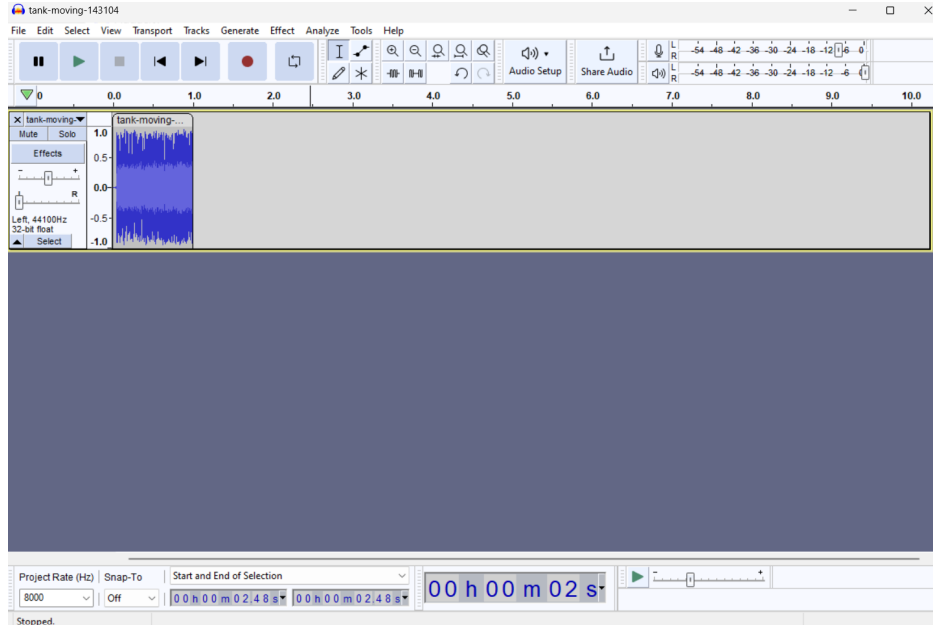


Figure 2.3a: Sample formatted in Audacity

The .mif files were used to prepopulate the on-chip memory ROMs to contain the music data. We modified the Qsys interface such that our audio samples would be properly fed into the WM8731 CODEC. The 3 main components involved were: the altera\_up\_avalon\_audio\_pll, the altera\_up\_avalon\_audio\_and\_video\_config, and the altera\_up\_avalon\_audio IPs. The altera\_up\_avalon\_audio\_pll acts as a clock divider. As the CODEC does not operate on our standard 50 MHz, the PLL is needed to create a 12.288 MHz clock frequency, using the 50 MHz clock as a reference. The altera\_up\_avalon\_audio\_and\_video\_config sets up our peripheral audio device – configures the CODEC – given our initialization arguments: left-justified data format, 16 bit length, etc. The altera\_up\_avalon\_audio facilitates a transfer of audio between our WM8731 CODEC and FPGA through right and left channels implemented as FIFOs. Together, these IP blocks gave us a data channel and clock prepared for the line out jack through the Wolfson WM8731 CODEC.

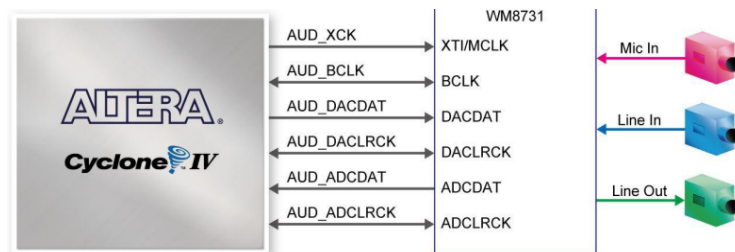


Figure 2.3b: Wolfson WM8731 I/O.

Use	Connections	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>		<b>clk_0</b>	Clock Source	<b>clk</b>	<b>exported</b>		
		clk_in	Clock Input	reset			
		clk_reset	Reset Input	clk_0			
		clk	Clock Output				
		clk_reset	Reset Output				
<input checked="" type="checkbox"/>		<b>hps_0</b>	Arria V/Cyclone V Hard Proce...				
		h2f_user1_clock	Clock Output	hps_0_h2...			
		memory	Conduit	hps_ddr3			
		hps_io	Conduit	hps			
		h2f_reset	Reset Output				
		h2f_axi_clock	Clock Input		clk_0		
		h2f_axi_master	AXI Master		[h2f_axi_...		
		h2f_axi_slave	AXI Slave		clk_0		
		h2f_lw_axi_slave	Clock Input		[h2f_axi_...		
		h2f_lw_axi_master	AXI Master		clk_0		
		h2f_irq0	Interrupt Receiver		[h2f_lw_a...		
		h2f_irq1	Interrupt Receiver			IRQ 0	IRQ 31
						IRQ 0	IRQ 31
<input checked="" type="checkbox"/>		<b>vga_ball_0</b>	VGA Ball				
		clock	Clock Input		clk_0		
		reset	Reset Input		[clock]		
		avalon_slave_0	Avalon Memory Mapped Slave		[clock]	# 0x0000_0000	0x0000_001f
		vga	Conduit	vga	[clock]		
		avalon_streamin...	Avalon Streaming Source		[clock]		
		avalon_streamin...	Avalon Streaming Source		[clock]		
		reset1	Reset Input		[clock]		
<input checked="" type="checkbox"/>		<b>jingle_sound</b>	On-Chip Memory (RAM or ROM...				
		clk1	Clock Input		clk_0		
		s1	Avalon Memory Mapped Slave		[clk1]	# 0x0009_0000	0x0009_7cff
		reset1	Reset Input		[clk1]		
<input checked="" type="checkbox"/>		<b>shoot_sound</b>	On-Chip Memory (RAM or ROM...				
		clk1	Clock Input		clk_0		
		s1	Avalon Memory Mapped Slave		[clk1]	# 0x0010_0000	0x0010_84cf
		reset1	Reset Input		[clk1]		
<input checked="" type="checkbox"/>		<b>crawl_sound</b>	On-Chip Memory (RAM or ROM...				
		clk1	Clock Input		clk_0		
		s1	Avalon Memory Mapped Slave		[clk1]	# 0x0011_0000	0x0011_464f
		reset1	Reset Input		[clk1]		
<input checked="" type="checkbox"/>		<b>explode_sound</b>	On-Chip Memory (RAM or ROM...				
		clk1	Clock Input		clk_0		
		s1	Avalon Memory Mapped Slave		[clk1]	# 0x0012_0000	0x0012_658f
		reset1	Reset Input		[clk1]		
<input checked="" type="checkbox"/>		<b>audio_pll_0</b>	Audio Clock for DE-series Boa...				
		ref_clk	Clock Input		clk_0		
		ref_reset	Reset Input				
		audio_clk	Clock Output	audio_pll_0_audio_...	audio_pll...		
		reset_source	Reset Output				
<input checked="" type="checkbox"/>		<b>audio_and_vid...</b>	Audio and Video Config				
		clk	Clock Input		clk_0		
		reset	Reset Input		[clk]		
		avalon_av_config...	Avalon Memory Mapped Slave		[clk]		
		external_interface	Conduit	audio_and_video_c...			
<input checked="" type="checkbox"/>		<b>audio_0</b>	Audio				
		clk	Clock Input		clk_0		
		reset	Reset Input		[clk]		
		avalon_left_chan...	Avalon Streaming Source		[clk]		
		avalon_right_chan...	Avalon Streaming Source		[clk]		
		avalon_left_chan...	Avalon Streaming Sink		[clk]		
		avalon_right_chan...	Avalon Streaming Sink		[clk]		
		external_interface	Conduit	audio_0_external_1...			

Figure 2.3c: Qsys audio connections.

In the figure above are our final Qsys connections that facilitate audio data transfer between our FPGA and WM8731 CODEC peripheral. vga\_ball\_0 has avalon\_streaming\_interfaces for both the left and right channels, with each of these interfaces having a ready, valid, and data signal. These signals are used to judge when the CODEC FIFOs are prepared to accept audio samples. The audio loop waits for the altera\_up\_avalon\_audio IP to send a HIGH ready signal and proceeds to count up to a threshold, slowing our data transfer to an intelligible rate. Once this threshold is met, valid signals go HIGH and depending on the game event, some audio sample is passed to the CODEC through the altera\_up\_avalon\_audio.



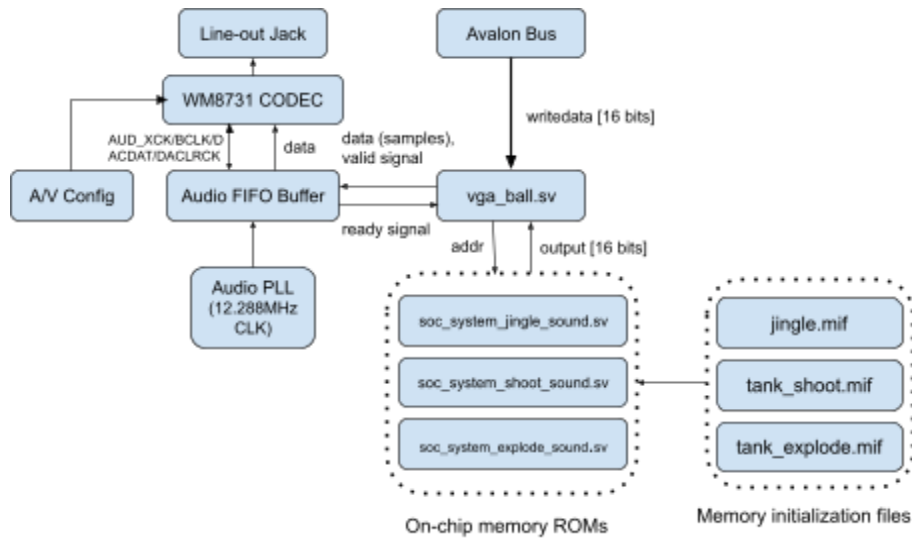




Figure 2.3d: Audio architecture.

## 2.4 Memory

The FPGA includes 4450 Kbits of embedded memory. The sprites and audio required for our project are shown in the table below. The audio requires the most memory. For the tanks, four images are needed for each of the possible directions the tanks can face (up, down, left, right). The maze is constructed out of a single wall image. To display the score, . The score is displayed independently for both players, so one set for each player gives a total of 6 images.

Name	Graphic	Size (bits)	# Required	Total Size (bytes)
Player 1 Tank		32 x 32	4	4096
Player 2 Tank		32 x 32	4	4096
Wall		32 x 32	1	1024
Map	N/A	N/A	N/A	900
Explosion		32 x 32	3	3072
Ending		32 x 32	2	2048
Numbers		16 x 16	5	1280

Score		32 x 32	5	5120
Stage		16 x 16	5	1280
		Memory Budget (bits)		183328

The value for each pixel in the sprite images was stored as a one byte hex value, resulting in the respective total sizes for each image in the table above.

Additionally, we need ROMs to store our audio files that play upon specific game events. These were sampled from .mp3 files at 8 kHz to become binary .wav files. Each sample is stored in memory as a signed 16-bit integer to be fed into our WM8731. For our implementation, we chose to use 3 game events to trigger audio: start, shoot, and explode. A jingle will play whenever a new game starts, and there will be booming noises upon shooting a bullet and tanks exploding.

Name	Samples	Total Size (bytes)
Jingle	15168	30336
Shoot	16099	32196
Explode	12109	24218
	Memory Budget (bits)	694000

In all, for our sprites we utilized 183 Kbits of memory, while for our audio we used 694 Kbits. This totals to 877 Kbits, which is less than the embedded memory in the FPGA.

## 3 Software

### 3.1 User Input

#### 3.1.1 Overview

Players interact with the game using a pair of iNNext game controllers, which can be seen below:



Figure 3.1: Our game controllers

The controllers are connected via USB and are identified by an `idProduct` of 17 and have just a single interface. Players can move the tanks up/down/left/right using the arrow keys on the controllers, and fire a bullet using the A button. Counter variables were used for each of the arrow keys and A button to slow down the input speed and to in effect debounce the switches so that a single physical press was only registered as a single press of the button on the software side. Both of the controller inputs are handled in a single loop as threads were not necessary to maintain fast and simultaneous input latency.

We modified the `controller.c` skeleton for our controller setup, changing the keyboard-opening function to return a structure that carries information about both of our controllers and altering the constraints such that it would only connect/open our desired controllers. With two connected controllers, we sequentially used `libusb_interrupt_transfer` to read their 7 byte protocol messages into another structure containing an attribute for each field. These structures are later processed in our game-loop to determine when a tank is moving, or shooting.

### 3.1.2 Communication Protocol

Each controller communicates using the following 7 byte protocol:

Constant	Constant	Constant	Left/Right	Up/Down	X, Y, A, B	Left_Bumper, Right_Bumper, Select, Start
----------	----------	----------	------------	---------	------------	--

There are three constant fields at the beginning of each controller packet, describing the protocol. Our controller is not identified by one of the libusb built in protocols, so it fills these fields with 255 in each. This corresponds to ‘protocol 0,’ whereas an identifiable protocol such as that of the keyboards might be ‘protocol 1.’

For the left/right field, it defaults to a 127 integer, which changes to 0 if left is pressed, or 255 if right is pressed. The up/down field works the same way, dropping to 0 if up is pressed; 255 if down is pressed. The X, Y, A, B field has a more complex representation of button presses:

Integer Value	Keys Pressed
15	None
31	X
47	A
63	A, X
79	B
95	B, X
111	A, B
127	A, B, X

Integer Value	Keys Pressed
143	Y
159	X, Y
175	A, Y
191	A, X, Y
207	B, Y
223	B, X, Y
239	A, B, Y
255	A, B, X, Y

The scheme is additive, where pressing different keys adds to the integer total. The Left\_Buffer, Right\_Buffer, Start, Select field operates in the same way, simply replacing the A, B, X, Y keys.

## 3.2 Game Logic

### 3.2.1 Overview

The userspace program to handle the game logic has three primary loops– 1) loop to restart the game, 2) loop to select the stage, 3) loop to play the game. Upon startup, the currently selected stage is displayed on the screen and the program listens for user input from controller 1. The stage number corresponds to which map will be selected, with there being 3 different playable maps. When the up arrow is pressed, the stage number increments and the player tanks displayed move in a little animation. The speed of the animation is dependent on the stage number selected and acts as an indication to the complexity of the selected map. When the down arrow is pressed, the stage number decrements. The user selects the current stage and starts the game by pressing the A button. Once the game starts, the players are free to move around the maze and fire bullets at the opponent's tank. Tanks and bullets cannot move through walls.

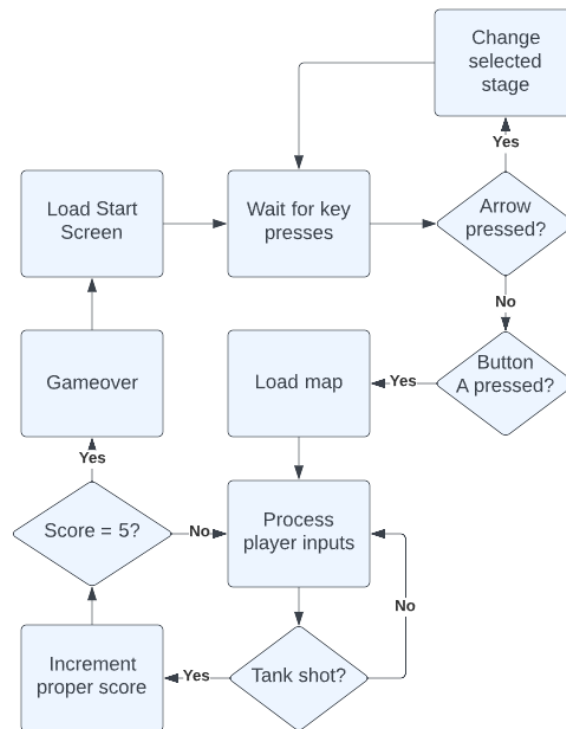


Figure 3.2.1: Flowchart of game logic

### 3.2.2 Tank Movement

Tank movement is discretized in increments to give the game a familiar feel and aid player movement around and through the block maze. We decided to keep the tank movement speed constant. Holding the arrow key moves the tank continually in the specified direction. When the player presses a different arrow key and the intended spot is vacant, the tank turns in that direction. The Player 1 tank is gold in color and the Player 2 tank is silver.

### 3.2.3 Collision Detection

Valid player movement is determined through collision detection with both the walls of the maze and the opponent tank. The coordinates of the tank are truncated into a 32 bit value for each x and y direction, yielding a 20x15 grid, and used to determine an overlap between tank and wall or tank and tank and prevent movement through.

### 3.2.4 Bullet Firing + Movement

Players can only fire one bullet at a time. The bullet for Player 1 matches the gold color of the Player 1 tank and the bullet for Player 2 matches the silver color for the Player 2 tank. Bullets move faster than tanks and fire in the current direction the tank is facing. Once the bullet hits a wall, it disappears, and the player is able to fire a new bullet. If the bullet hits the opponent's tank, the tank explodes and the bullet disappears. Bullet collision is detected in the same manner as tank and wall collisions as described above. Separate threads are used for each of the players' bullets so that the bullets can fire and move simultaneously with the tanks themselves, which are handled by the main thread of the program.

### 3.2.5 End Game

Successfully shooting the opponent's tank gives the player 100 points, indicated by the score at the top of the screen. Player 1's score is on the left side and Player 2's score is on the right. The first player to land 5 hits on the opponent and score 500 points wins the game, upon which the "gameover" graphic is displayed on top of the maze and the game resets back to the stage selection screen for replay.

### 3.3 Avalon Bus Interface

Communication from the software to the hardware is done through the device driver. Ioctl16 writes of 16 bits are made to 10 different registers to send data information from the software for handling by the hardware graphics. The Avalon bus interface is specified below:

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Details	
00	Stage num		Player 1 Score			Player 2 Score			Explosion		End- ng En	Expl- osion En	P1 Score En	Wall En	P2 Score En			Game information
02	X Position							Y Position							Map num	Explosion, Map		
04	X Position							Y Position									Player 1 Tank	
06	Direction																	
08	X Position							Y Position									Player 2 Tank	
10	Direction																	
12	X Position							Y Position									Player 1 Bullet	
14	Bullet En																	
16	X Position							Y Position									Player 2 Bullet	
18	Bullet En																	

There are a total of 10 2-byte registers that we use, performing writes of 16 bits to pass data from the software to the hardware. Address 00 is for data such as the stage num, player scores, explosion graphic, and various enable bits to turn on the respective graphics. Address 02 is the coordinates for the explosion and the selected map number. Addresses 04 and 06 are for player 1 tank information including the coordinates and direction, and addresses 08 and 10 are for the player 2 tank. Addresses 12 and 14 are for the player 1 bullet coordinates and enable, and addresses 16 and 18 for the player 2 bullet.

## 4 Discussion

### 4.1 Challenges

- To store the sprites and audio in memory, we had to convert the images and samples into a format that would allow us to initialize the ROMs and store the required data in memory. We had to use a publicly available GitHub repo along with custom Python scripts to convert the images into .mem formats, scale them up, then convert them to .mif format. Also, the images had to be altered so that we did not exceed the allocated color palette size. Another GitHub repo was used to convert the .wav audio files into .mif format as well.
- The input from the controllers had a lot of switch bounce in the sense that one physical press of a button doesn't translate to one press seen in the software. And so, counter variables had to be used to limit the rate at which input was taken from the controllers so that button presses translated into movement and triggers as the user would expect.
- The bullets and tanks needed to be independent as the user should be able to still move the tank after firing a bullet. And so, the bullet firing and movement had to be handled in separate threads so that the main program thread could continue handling user input and the tank movement. As a result, certain variables had to be made global to allow access by the different threads and requisite information passed as a struct to the thread. Also, upon a tank explosion and the end of a round, the threads had to be canceled and stopped to reset the map.
- Another challenge we faced was having the sprites phase through each other. The issue was with the collision detection between the tank/bullet and the wall sprites. We refined the way we detected collisions by truncating the coordinates of the tank/bullet to 32 bit values (as the walls were 32x32 pixels) and then checking for overlap.
- We initially faced issues troubleshooting the audio where it wouldn't play. There wasn't much documentation available to figure out how to connect the pins in the top-level sv file to output the audio. After we were able to figure this step out, we were able to work on getting the audio to play and then to have it play at specific triggers.

### 4.2 Lessons Learned + Advice

- A lesson we learned was to try components out in stages. Especially with setting up the hardware graphics, ensuring things were done and tested in stages (such as one new sprite ROM at a time) ensured that if any debugging was required, then the problem could be more easily identified.
- Another lesson in regards to the software was writing the game logic carefully and being sure that each function/new behavior was implemented properly and considered with respect to other



existing behavior. This was important to ensure that there were no bugs in our game execution and that the logic worked as intended.

- Our advice is to focus on getting the smaller pieces working and then building everything together. We first started by setting up one ROM and getting that sprite to display as a static image on the display. Once we had that, then we tried moving it through the user program and device driver, before moving on and adding more sprites.
- Additionally, using available tools such as public GitHub repos (to convert .png images and .wav files into .mif format) was a good starting point and helped us in getting the requisite data we needed to build or project components.

### 4.3 Who did what section?

- Ganesan led the sprite graphics and game logic
- Quinn led the audio and controls
- Ana Maria led the documentation and helped immensely with the audio/game logic and testing
- Of course, we all helped each other out in all aspects as well

## 5 References

- [1] <https://projectf.io/posts/hardware-sprites/>
- [2] <https://github.com/projf/fpgatools>
- [3] <https://github.com/0x60/385-audio-tools>
- [4] <https://htmlcolorcodes.com/>
- [5] [Computer Laboratory – Course pages 2016–17: ECAD and Architecture Practical Classes – Tutorial \(cam.ac.uk\)](#)

# 6 Code

## 1. vga\_ball.sv

```
/*
 * Avalon memory-mapped peripheral that generates Battle City tank game
visuals
 *
 * Contributors:
 *
 * Quinn Booth
 * Columbia University
 *
 * Ganesan Narayanan
 * Columbia University
 *
 * Ana Maria Rodriguez
 * Columbia University
 *
 * Skeleton by:
 * Stephen A. Edwards
 * Columbia University
 */

module vga_ball(input logic          clk,
               input logic          reset,
               input logic [15:0]   writedata,
               input logic          write,
               input                chipselect,
               input logic [3:0]    address,

               input L_READY,
               input R_READY,
               output logic [15:0] L_DATA,
               output logic [15:0] R_DATA,
               output logic L_VALID,
               output logic R_VALID,

               output logic [7:0]   VGA_R, VGA_G, VGA_B,
               output logic        VGA_CLK, VGA_HS, VGA_VS,
                                   VGA_BLANK_n,
               output logic        VGA_SYNC_n);

logic [10:0]   hcount;
logic [9:0]   vcount;

logic [7:0]   background_r, background_g, background_b;

logic [15:0]  wall, misc, p1_tankl, p1_tankd, p2_tankl, p2_tankd,
p1_bulletl, p1_bulletd, p2_bulletl, p2_bulletd;
```

```

    vga_counters counters(.clk50(clk), .*);

////////// AUDIO ////////////////////////////////////////////

    logic [13:0] jingle_address;
    logic [15:0] jingle_data;

    soc_system_jingle_sound(.address(jingle_address),.clk(clk),.clken(1),.reset_req(0),.readdata(jingle_data));

    logic [13:0] explode_address;
    logic [15:0] explode_data;

    soc_system_explode_sound(.address(explode_address),.clk(clk),.clken(1),.reset_req(0),.readdata(explode_data));

    logic [13:0] crawl_address;
    logic [15:0] crawl_data;

    soc_system_crawl_sound(.address(crawl_address),.clk(clk),.clken(1),.reset_req(0),.readdata(crawl_data));

    logic [14:0] shoot_address;
    logic [15:0] shoot_data;

    soc_system_shoot_sound(.address(shoot_address),.clk(clk),.clken(1),.reset_req(0),.readdata(shoot_data));

    reg [11:0] counter;
    logic playing_jingle;
    logic playing_explode;
    logic playing_crawl;
    logic playing_bullet1;
    logic playing_bullet2;
    logic playing_initial_jingle;

    logic jingle_ready;
    logic explode_ready;
    logic bullet1_ready;
    logic bullet2_ready;

    always_ff @(posedge clk) begin

        if (reset) begin

            counter <= 0;
            L_VALID <= 0;
            R_VALID <= 0;
            playing_jingle <= 0;
            playing_explode <= 0;
            playing_crawl <= 0;
            playing_bullet1 <= 0;
            playing_bullet2 <= 0;
        end
    end

```

```

playing_initial_jingle <= 1;
jingle_address <= 0;
explode_address <= 0;
crawl_address <= 0;
shoot_address <= 0;
jingle_ready <= 1;
explode_ready <= 1;
bullet1_ready <= 1;
bullet2_ready <= 1;

end
else if (L_READY == 1 && R_READY == 1 && counter < 3125) begin

    counter <= counter + 1;
    L_VALID <= 0;
    R_VALID <= 0;

end
else if (L_READY == 1 && R_READY == 1 && counter >= 3125) begin

    counter <= 0;
    L_VALID <= 1;
    R_VALID <= 1;

    // Play the opening jingle on boot
    if (playing_initial_jingle == 1) begin

        if (jingle_address > 15000) begin
            jingle_address <= 0;
            playing_initial_jingle <= 0;
        end
        else begin
            jingle_address <= jingle_address + 1;
        end
        L_DATA <= jingle_data;
        R_DATA <= jingle_data;

    end

    // After the initial jingle, handle audio on game events
    else begin

        // Setting Flags

        if (wall_on == 1'b1 && jingle_ready == 1'b1) begin

            if (playing_explode == 0 && playing_bullet1 == 0 &&
                playing_bullet2 == 0) begin

                playing_jingle <= 1;
                playing_explode <= 0;
                playing_bullet1 <= 0;
                playing_bullet2 <= 0;
            end
        end
    end
end

```

```

    jingle_ready <= 0;

end

end
else if (explosion_on == 1'b1 && explode_ready == 1'b1) begin

    if (playing_jingle == 0 && playing_bullet1 == 0 &&
playing_bullet2 == 0) begin

        playing_jingle <= 0;
        playing_explode <= 1;
        playing_bullet1 <= 0;
        playing_bullet2 <= 0;

        explode_ready <= 0;

    end

end

end
else if (bullet1_on == 1'b1 && bullet1_ready == 1'b1) begin

    if (playing_jingle == 0 && playing_explode == 0 &&
playing_bullet2 == 0) begin

        playing_jingle <= 0;
        playing_explode <= 0;
        playing_bullet1 <= 1;
        playing_bullet2 <= 0;

        bullet1_ready <= 0;

    end

end

end
else if (bullet2_on == 1'b1 && bullet2_ready == 1'b1) begin

    if (playing_jingle == 0 && playing_bullet1 == 0 &&
playing_explode == 0) begin

        playing_jingle <= 0;
        playing_explode <= 0;
        playing_bullet1 <= 0;
        playing_bullet2 <= 1;

        bullet2_ready <= 0;

    end

end

end

// These _on flags represent when in game events are "on" or
occurring

```

```

    // Only allow the audio to play again once they drop to 0, so that
    there is 1 audio per event

    if (wall_on == 1'b0) begin
        jingle_ready <= 1;
    end

    if (explosion_on == 1'b0) begin
        explode_ready <= 1;
    end

    if (bullet1_on == 1'b0) begin
        bullet1_ready <= 1;
    end

    if (bullet2_on == 1'b0) begin
        bullet2_ready <= 1;
    end

    // Playing Audio

    if (playing_jingle == 1) begin
        if (jingle_address > 15000) begin
            jingle_address <= 0;
            playing_jingle <= 0;
        end
        else begin
            jingle_address <= jingle_address + 1;
        end
        L_DATA <= jingle_data;
        R_DATA <= jingle_data;
    end

    else if (playing_explode == 1) begin
        if (explode_address > 15000) begin
            explode_address <= 0;
            playing_explode <= 0;
        end
        else begin
            explode_address <= explode_address + 1;
        end
        L_DATA <= explode_data;
        R_DATA <= explode_data;
    end

    else if (playing_bullet1 == 1) begin

```

```

        if (shoot_address > 15000) begin
            shoot_address <= 0;
            playing_bullet1 <= 0;
        end
        else begin
            shoot_address <= shoot_address + 1;
        end
        L_DATA <= shoot_data;
        R_DATA <= shoot_data;
    end
    else if (playing_bullet2 == 1) begin
        if (shoot_address > 15000) begin
            shoot_address <= 0;
            playing_bullet2 <= 0;
        end
        else begin
            shoot_address <= shoot_address + 1;
        end
        L_DATA <= shoot_data;
        R_DATA <= shoot_data;
    end
end

end

else begin

    L_VALID <= 0;
    R_VALID <= 0;

end

end

////////// AUDIO ////////////////////////////////////////

always_ff @(posedge clk)
    if (reset) begin
        background_r <= 8'h00;
        background_g <= 8'h00;
        background_b <= 8'h00;
        // set any default values needed for startup
    end
    else if (chipselct && write)
        case (address)
            4'h0 : wall <= writedata;
            4'h1 : misc <= writedata;
            4'h2 : p1_tankl <= writedata;
            4'h3 : p1_tankd <= writedata;
            4'h4 : p2_tankl <= writedata;
            4'h5 : p2_tankd <= writedata;
            4'h6 : p1_bulletl <= writedata;
            4'h7 : p1_bulletd <= writedata;
            4'h8 : p2_bulletl <= writedata;
        end
    end

```



```

        4'h9 : p2_bulletd <= writedata;
    endcase

    logic [11:0] p1tank_address;
    logic [7:0] p1tank_output;
    soc_system_p1tank_unit
p1tank_unit(.address(p1tank_address),.clk(clk),.clken(1),.reset_req(0),.re
addata(p1tank_output));
    logic [1:0] p1tank_en;

    logic [9:0] p1tank_x;
    logic [9:0] p1tank_y;
    logic [1:0] p1tank_dir;

    logic [11:0] p2tank_address;
    logic [7:0] p2tank_output;
    soc_system_p2tank_unit
p2tank_unit(.address(p2tank_address),.clk(clk),.clken(1),.reset_req(0),.re
addata(p2tank_output));
    logic [1:0] p2tank_en;

    logic [9:0] p2tank_x;
    logic [9:0] p2tank_y;
    logic [1:0] p2tank_dir;

    logic [9:0] map_address;
    logic [7:0] map_output;
    soc_system_map_unit
map_unit(.address(map_address),.clk(clk),.clken(1),.reset_req(0),.readdata
(map_output));
    logic [1:0] map_en;

    logic [1:0] map_num;

    logic [9:0] wall_address;
    logic [7:0] wall_output;
    soc_system_wall_unit
wall_unit(.address(wall_address),.clk(clk),.clken(1),.reset_req(0),.readda
ta(wall_output));
    logic [1:0] wall_en;
    logic wall_on;

    logic [4:0] tile32_x;
    logic [4:0] tile32_y;

    logic [12:0] score_address;
    logic [7:0] score_output;
    soc_system_score_unit
score_unit(.address(score_address),.clk(clk),.clken(1),.reset_req(0),.read
data(score_output));
    logic [1:0] score_en;
    logic score1_on;
    logic score2_on;

```

```

    logic [10:0] stage_address;
    logic [7:0] stage_output;
    soc_system_stage_unit
stage_unit(.address(stage_address),.clk(clk),.clken(1),.reset_req(0),.read
data(stage_output));
    logic [1:0] stage_en;

    logic [5:0] tile16_x;
    logic [5:0] tile16_y;

    logic [10:0] num_address;
    logic [7:0] num_output;
    soc_system_num_unit
num_unit(.address(num_address),.clk(clk),.clken(1),.reset_req(0),.readdata
(num_output));
    logic [1:0] num_en;

    logic [1:0] stage_num;

    logic [2:0] p1_score;
    logic [2:0] p2_score;

    logic [9:0] p1bullet_x;
    logic [9:0] p1bullet_y;
    logic [9:0] p2bullet_x;
    logic [9:0] p2bullet_y;

    logic [9:0] bullet1_xdif;
    logic [9:0] bullet1_ydif;
    logic [9:0] bullet2_xdif;
    logic [9:0] bullet2_ydif;

    logic [1:0] p1bullet_en;
    logic [1:0] p2bullet_en;

    logic bullet1_on;
    logic bullet2_on;

    logic [10:0] ending_address;
    logic [7:0] ending_output;
    soc_system_ending_unit
ending_unit(.address(ending_address),.clk(clk),.clken(1),.reset_req(0),.re
addata(ending_output));
    logic [1:0] ending_en;
    logic ending_on;

    logic [11:0] explosion_address;
    logic [7:0] explosion_output;
    soc_system_explosion_unit
explosion_unit(.address(explosion_address),.clk(clk),.clken(1),.reset_req(
0),.readdata(explosion_output));
    logic [1:0] explosion_en;
    logic explosion_on;

```

```

logic [9:0] explosion_x;
logic [9:0] explosion_y;
logic [1:0] explosion_num;

// ptank_dir:
// 2'b0 --> up
// 2'b1 --> down
// 2'b2 --> left
// 2'b3 --> right

always_comb begin

    p1tank_x = p1_tankl[15:8] << 2;
    p1tank_y = p1_tankl[7:0] << 2;
    p1tank_dir = p1_tankd[15:14];

    p2tank_x = p2_tankl[15:8] << 2;
    p2tank_y = p2_tankl[7:0] << 2;
    p2tank_dir = p2_tankd[15:14];

    tile32_x = hcount[10:1] >> 5;
    tile32_y = vcount[9:0] >> 5;

    tile16_x = hcount[10:1] >> 4;
    tile16_y = vcount[9:0] >> 4;

    stage_num = wall[15:14];

    p1_score = wall[13:11];
    p2_score = wall[10:8];

    p1bullet_x = p1_bulletl[15:8] << 2;
    p1bullet_y = p1_bulletl[7:0] << 2;

    p2bullet_x = p2_bulletl[15:8] << 2;
    p2bullet_y = p2_bulletl[7:0] << 2;

    bullet1_xdif = (p1bullet_x > hcount[10:1]) ? (p1bullet_x -
hcount[10:1]) : (hcount[10:1] - p1bullet_x);
    bullet1_ydif = (p1bullet_y > vcount[9:0]) ? (p1bullet_y -
vcount[9:0]) : (vcount[9:0] - p1bullet_y);

    bullet2_xdif = (p2bullet_x > hcount[10:1]) ? (p2bullet_x -
hcount[10:1]) : (hcount[10:1] - p2bullet_x);
    bullet2_ydif = (p2bullet_y > vcount[9:0]) ? (p2bullet_y -
vcount[9:0]) : (vcount[9:0] - p2bullet_y);

    explosion_x = misc[15:9] << 3;
    explosion_y = misc[8:2] << 3;

    explosion_num = wall[7:6];

    ending_on = wall[5];

```

```

explosion_on = wall[4];
score1_on = wall[3];
wall_on = wall[2];
score2_on = wall[1];

bullet1_on = p1_bulletd[15];
bullet2_on = p2_bulletd[15];

map_num = misc[1:0];

map_en = 2'b1;

if (map_num == 2'b00)
map_address = tile32_x + tile32_y * 20;
else if (map_num == 2'b01)
map_address = tile32_x + tile32_y * 20 + 300;
else
map_address = tile32_x + tile32_y * 20 + 600;

end

always_ff @(posedge clk) begin

if (hcount[10:1] >= p1tank_x && hcount[10:1] <= (p1tank_x + 10'd31)
&& vcount[9:0] >= p1tank_y && vcount[9:0] <= (p1tank_y + 10'd31) ) begin

p1tank_en <= 2'b1;

case(p1tank_dir)
2'b00 : p1tank_address <= hcount[10:1] - p1tank_x + (vcount[9:0] -
p1tank_y) * 32;
2'b01 : p1tank_address <= hcount[10:1] - p1tank_x + (vcount[9:0]
- p1tank_y) * 32 + 1024;
2'b10 : p1tank_address <= hcount[10:1] - p1tank_x + (vcount[9:0]
- p1tank_y) * 32 + 2048;
2'b11 : p1tank_address <= hcount[10:1] - p1tank_x + (vcount[9:0]
- p1tank_y) * 32 + 3072;
endcase

end

else begin

p1tank_en <= 2'b0;

end

end

always_ff @(posedge clk) begin

if (hcount[10:1] >= p2tank_x && hcount[10:1] <= (p2tank_x + 10'd31)
&& vcount[9:0] >= p2tank_y && vcount[9:0] <= (p2tank_y + 10'd31) ) begin

```

```

    p2tank_en <= 2'b1;

    case(p2tank_dir)
        2'b00 : p2tank_address <= hcount[10:1] - p2tank_x + (vcount[9:0] -
p2tank_y) * 32;
        2'b01 : p2tank_address <= hcount[10:1] - p2tank_x + (vcount[9:0]
- p2tank_y) * 32 + 1024;
        2'b10 : p2tank_address <= hcount[10:1] - p2tank_x + (vcount[9:0]
- p2tank_y) * 32 + 2048;
        2'b11 : p2tank_address <= hcount[10:1] - p2tank_x + (vcount[9:0]
- p2tank_y) * 32 + 3072;
    endcase

    end

    else begin

        p2tank_en <= 2'b0;

    end

end

always_ff @(posedge clk) begin

    if (hcount[10:1] >= explosion_x && hcount[10:1] <= (explosion_x +
10'd31) && vcount[9:0] >= explosion_y && vcount[9:0] <= (explosion_y +
10'd31) && explosion_on == 1'b1) begin

        explosion_en <= 2'b1;

        case(explosion_num)
            2'b00 : explosion_address <= hcount[10:1] - explosion_x +
(vcount[9:0] - explosion_y) * 32;
            2'b01 : explosion_address <= hcount[10:1] - explosion_x +
(vcount[9:0] - explosion_y) * 32 + 1024;
            2'b10 : explosion_address <= hcount[10:1] - explosion_x +
(vcount[9:0] - explosion_y) * 32 + 2048;
        endcase

        end

        else begin

            explosion_en <= 2'b0;

        end

    end

always_ff @(posedge clk) begin

```

```

        if (tile32_x >= 6'd9 && tile32_x <= 6'd10 && tile32_y == 6'd7 &&
ending_on == 1'b1) begin

            ending_en <= 2'b1;
            ending_address <= hcount[5:1] + vcount[4:0] * 32 + (tile32_x -
6'd9) * 1024;

        end

        else begin

            ending_en <= 2'b0;

        end

    end

    always_ff @(posedge clk) begin

        if (tile16_x >= 6'd17 && tile16_x <= 6'd21 && tile16_y == 6'd0)
begin

            stage_en <= 2'b1;
            stage_address <= hcount[4:1] + vcount[3:0] * 16 + (tile16_x -
6'd17) * 256;

        end

        else begin

            stage_en <= 2'b0;

        end

    end

    always_ff @(posedge clk) begin

        if (tile16_x == 6'd22 && tile16_y == 6'd0) begin

            num_en <= 2'b1;

            case(stage_num)
                2'b00 : num_address <= hcount[4:1] + vcount[3:0] * 16 + 256;
                2'b01 : num_address <= hcount[4:1] + vcount[3:0] * 16 + 512;
                2'b10 : num_address <= hcount[4:1] + vcount[3:0] * 16 + 768;
                //2'b11 : num_address <= hcount[4:1] + vcount[3:0] * 16 + 1024;
            endcase

        end

    end

```

```

else begin

    num_en <= 2'b0;

end

end

always_ff @(posedge clk) begin

    if (tile32_x == 5'd2 && tile32_y == 5'd0 && score1_on == 1'b1) begin

        score_en <= 2'b1;

        case(p1_score)
            3'b000 : score_address <= hcount[5:1] + vcount[4:0] * 32;
            3'b001 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 1024;
            3'b010 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 2048;
            3'b011 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 3072;
            3'b100 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 4096;
        endcase

    end

    else if (tile32_x == 5'd17 && tile32_y == 5'd0 && score2_on == 1'b1)
begin

        score_en <= 2'b1;

        case(p2_score)
            3'b000 : score_address <= hcount[5:1] + vcount[4:0] * 32;
            3'b001 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 1024;
            3'b010 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 2048;
            3'b011 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 3072;
            3'b100 : score_address <= hcount[5:1] + vcount[4:0] * 32 + 4096;
        endcase

    end

    else begin

        score_en <= 2'b0;

    end

end

always_ff @(posedge clk) begin

    if ( (bullet1_xdif < 5) && (bullet1_ydif < 5) && bullet1_on == 1'b1)
begin

```

```

        p1bullet_en <= 2'b1;
    end

    else begin

        p1bullet_en <= 2'b0;
    end

end

always_ff @(posedge clk) begin

    if ( (bullet2_xdif < 5) && (bullet2_ydif < 5) && bullet2_on == 1'b1)
begin

        p2bullet_en <= 2'b1;

    end

    else begin

        p2bullet_en <= 2'b0;
    end

end

/*
always_ff @(posedge clk) begin

    map_en <= 2'b1;

    case(map_num)
        2'b00 : map_address <= tile32_x + tile32_y * 20;
        2'b01 : map_address <= tile32_x + tile32_y * 20 + 300;
        2'b10 : map_address <= tile32_x + tile32_y * 20 + 600;
    endcase

end
*/

always_ff @(posedge clk) begin

    if (map_en && map_output && wall_on == 1'b1) begin

        wall_en <= 2'b1;
        wall_address <= hcount[5:1] + vcount[4:0] * 32;

    end

    else begin

```



```

    wall_en <= 2'b0;

end

end

// This is where the colors of the screen are being set
always_comb begin
    {VGA_R, VGA_G, VGA_B} = {background_r, background_g, background_b};
    if (VGA_BLANK_n ) begin
if (explosion_en) begin
    case (explosion_output)
        8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
        8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
        8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
        8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
        8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
        8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
        8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
        8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
        8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
        8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
        8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
        8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
        8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
        8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
        8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
        8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
    endcase
end
    else if (ending_en) begin
        case (ending_output)
            8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
            8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
            8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
            8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
            8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
            8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
            8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
            8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
            8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
            8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
            8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
            8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
            8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
            8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
            8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
            8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
        endcase
    end
    else if (pltank_en) begin
        case (pltank_output)
            8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};

```

```

8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
endcase
end
else if (p2tank_en) begin
  case (p2tank_output)
    8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
    8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
    8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
    8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
    8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
    8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
    8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
    8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
    8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
    8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
    8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
    8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
    8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
    8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
    8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
    8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
  endcase
end
else if (wall_en) begin
  case (wall_output)
    8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
    8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
    8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
    8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
    8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
    8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
    8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
    8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
    8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
    8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
    8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
    8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
    8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
    8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
    8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
  endcase
end

```

```

    8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
endcase
end
else if (stage_en) begin
    case (stage_output)
        8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
        8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
        8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
        8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
        8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
        8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
        8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
        8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
        8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
        8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
        8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
        8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
        8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
        8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
        8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
        8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
    endcase
end
else if (num_en) begin
    case (num_output)
        8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
        8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
        8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
        8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
        8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
        8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
        8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
        8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
        8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
        8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
        8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
        8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
        8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
        8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
        8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
        8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
    endcase
end
else if (score_en) begin
    case (score_output)
        8'h00 : {VGA_R, VGA_G, VGA_B} = {8'hf0, 8'hf0, 8'hf0};
        8'h01 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'ha0, 8'ha0};
        8'h02 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'hb0};
        8'h03 : {VGA_R, VGA_G, VGA_B} = {8'ha0, 8'ha0, 8'ha0};
        8'h04 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h30, 8'h20};
        8'h05 : {VGA_R, VGA_G, VGA_B} = {8'hb0, 8'h20, 8'h20};
        8'h06 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'he0, 8'h90};
        8'h07 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h20};
        8'h08 : {VGA_R, VGA_G, VGA_B} = {8'he0, 8'h90, 8'h10};
    endcase
end

```

```

        8'h09 : {VGA_R, VGA_G, VGA_B} = {8'h90, 8'h40, 8'h00};
        8'h0a : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h60};
        8'h0b : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h60, 8'h00};
        8'h0c : {VGA_R, VGA_G, VGA_B} = {8'h60, 8'h00, 8'h00};
        8'h0d : {VGA_R, VGA_G, VGA_B} = {8'h50, 8'h00, 8'h70};
        8'h0e : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h40, 8'h40};
        8'h0f : {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
    endcase
end
else if (p1bullet_en) begin
    {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hf6, 8'h33};
end
else if (p2bullet_en) begin
    {VGA_R, VGA_G, VGA_B} = {8'hd8, 8'hd8, 8'hd8};
end
end
end
end

endmodule

module vga_counters(
    input logic      clk50, reset,
    output logic [10:0] hcount, // hcount[10:1] is pixel column
    output logic [9:0] vcount, // vcount[9:0] is pixel row
    output logic      VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0          1279          1599 0
 *
 * _____|_____ Video _____|_____ Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *
 * |_____|_____ VGA_HS _____|_____|_____
 */
// Parameters for hcount
parameter HACTIVE      = 11'd 1280,
          HFRONT_PORCH = 11'd 32,
          HSYNC        = 11'd 192,
          HBACK_PORCH  = 11'd 96,
          HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                        HBACK_PORCH; // 1600

// Parameters for vcount
parameter VACTIVE      = 10'd 480,
          VFRONT_PORCH = 10'd 10,
          VSYNC        = 10'd 2,
          VBACK_PORCH  = 10'd 33,
          VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                        VBACK_PORCH; // 525

```

```

logic endOfLine;

always_ff @(posedge clk50 or posedge reset)
    if (reset)          hcount <= 0;
    else if (endOfLine) hcount <= 0;
    else                hcount <= hcount + 11'd 1;

assign endOfLine = hcount == HTOTAL - 1;

logic endOfField;

always_ff @(posedge clk50 or posedge reset)
    if (reset)          vcount <= 0;
    else if (endOfLine)
        if (endOfField) vcount <= 0;
        else            vcount <= vcount + 10'd 1;

assign endOfField = vcount == VTOTAL - 1;

// Horizontal sync: from 0x520 to 0x5DF (0x57F)
// 101 0010 0000 to 101 1101 1111
assign VGA_HS = !( (hcount[10:8] == 3'b101) &
    !(hcount[7:5] == 3'b111));
assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal;
unused

// Horizontal active: 0 to 1279      Vertical active: 0 to 479
// 101 0000 0000 1280      01 1110 0000 480
// 110 0011 1111 1599      10 0000 1100 524
assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
    !( vcount[9] | (vcount[8:5] == 4'b1111) );

/* VGA_CLK is 25 MHz
*
* clk50    __|__|__|__|__|__|
*
*
* hcount[0]__|_____|_____|_____|
*/
assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule

```

## 2. vga\_ball.h

```

#ifndef _VGA BALL_H
#define _VGA BALL_H

#include <linux/ioctl.h>

```

```

typedef struct {
    unsigned short wall, misc, p1_tankl, p1_tankd, p2_tankl, p2_tankd,
    p1_bulletl, p1_bulletd, p2_bulletl, p2_bulletd;
} vga_ball_color_t;

typedef struct {
    vga_ball_color_t background;
} vga_ball_arg_t;

#define VGA BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA BALL_WRITE_BACKGROUND _IOW(VGA BALL_MAGIC, 1, vga_ball_arg_t
*)
#define VGA BALL_READ_BACKGROUND _IOR(VGA BALL_MAGIC, 2, vga_ball_arg_t
*)

#endif

```

### 3. vga\_ball.c

```

/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>

```

```

#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

#define DRIVER_NAME "vga_ball"

/* Device registers */
#define BG_WALL(x) (x)
#define BG_MISC(x) ((x)+2)
#define P1_TANKL(x) ((x)+4)
#define P1_TANKD(x) ((x)+6)
#define P2_TANKL(x) ((x)+8)
#define P2_TANKD(x) ((x)+10)
#define P1_BULLETL(x) ((x)+12)
#define P1_BULLETD(x) ((x)+14)
#define P2_BULLETL(x) ((x)+16)
#define P2_BULLETD(x) ((x)+18)

/*
 * Information about our device
 */
struct vga_ball_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory
 */
    vga_ball_color_t background;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_background(vga_ball_color_t *background)
{
    iowritel6(background->wall, BG_WALL(dev.virtbase) );
    iowritel6(background->misc, BG_MISC(dev.virtbase) );
    iowritel6(background->p1_tankl, P1_TANKL(dev.virtbase) );
    iowritel6(background->p1_tankd, P1_TANKD(dev.virtbase) );
    iowritel6(background->p2_tankl, P2_TANKL(dev.virtbase) );
    iowritel6(background->p2_tankd, P2_TANKD(dev.virtbase) );
    iowritel6(background->p1_bulletl, P1_BULLETL(dev.virtbase) );
    iowritel6(background->p1_bulletd, P1_BULLETD(dev.virtbase) );
    iowritel6(background->p2_bulletl, P2_BULLETL(dev.virtbase) );
    iowritel6(background->p2_bulletd, P2_BULLETD(dev.virtbase) );

    dev.background = *background;
}

/*

```

```

* Handle ioctl() calls from userspace:
* Read or write the segments on single digits.
* Note extensive error checking of arguments
*/
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long
arg)
{
    vga_ball_arg_t vla;

    switch (cmd) {
    case VGA BALL WRITE BACKGROUND:
        if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
            sizeof(vga_ball_arg_t)))
            return -EACCES;
        write_background(&vla.background);
        break;

    case VGA BALL READ BACKGROUND:
        vla.background = dev.background;
        if (copy_to_user((vga_ball_arg_t *) arg, &vla,
            sizeof(vga_ball_arg_t)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char
dev */
static struct miscdevice vga_ball_misc_device = {
    .minor          = MISC_DYNAMIC_MINOR,
    .name           = DRIVER_NAME,
    .fops           = &vga_ball_fops,
};

/*
* Initialization code: get resources (registers) and display
* a welcome message
*/
static int __init vga_ball_probe(struct platform_device *pdev)
{
    vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
    int ret;

```



```

/* Register ourselves as a misc device: creates /dev/vga_ball */
ret = misc_register(&vga_ball_misc_device);

/* Get the address of our registers from the device tree */
ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
if (ret) {
    ret = -ENOENT;
    goto out_deregister;
}

/* Make sure we can use these registers */
if (request_mem_region(dev.res.start, resource_size(&dev.res),
    DRIVER_NAME) == NULL) {
    ret = -EBUSY;
    goto out_deregister;
}

/* Arrange access to our registers */
dev.virtbase = of_iomap(pdev->dev.of_node, 0);
if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
}

/* Set an initial color */
write_background(&beige);

return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&vga_ball_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&vga_ball_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
    { .compatible = "csee4840,vga_ball-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

```

```

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_ball_driver = {
    .driver = {
        .name    = DRIVER_NAME,
        .owner   = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_ball_of_match),
    },
    .remove = __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
    platform_driver_unregister(&vga_ball_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");

```

#### 4. hello.c

```

/*
 * Tanks Game userspace program that runs the game logic and
 * communicates with the vga_ball device driver through ioctl's
 *
 * Contributors:
 *
 * Quinn Booth
 * Columbia University
 *
 * Ganesan Narayanan
 * Columbia University
 *
 * Ana Maria Rodriguez
 * Columbia University
 *
 * Stephen A. Edwards
 * Columbia University
 */

```





```

unsigned int tile32_y = v_coords >> 3;

if (map_num == 0) {
    if (map[tile32_x + tile32_y * 20] == 1)
        return 1;
}
else if (map_num == 1) {
    if (map2[tile32_x + tile32_y * 20] == 1)
        return 1;
}
else if (map_num == 2) {
    if (map3[tile32_x + tile32_y * 20] == 1)
        return 1;
}

return 0;
}

int check_collision_tank(unsigned short coords, unsigned short coords2)
{
    // returns 1 if collision

    // convert to tile

    unsigned char h_coords = coords >> 8;
    unsigned char v_coords = coords;

    unsigned int tile32_x = h_coords >> 3;
    unsigned int tile32_y = v_coords >> 3;

    unsigned char h_coords2 = coords2 >> 8;
    unsigned char v_coords2 = coords2;

    unsigned int tile32_x2 = h_coords2 >> 3;
    unsigned int tile32_y2 = v_coords2 >> 3;

    if (tile32_x == tile32_x2 && tile32_y == tile32_y2)
        return 1;

    return 0;
}

void *fire_bullet_p1(void *args) {

    p2_hit = 0;

    struct bullet_args *bullet_info = (struct bullet_args *) args;
    unsigned short cur_coords = bullet_info->cur_coords;
    int dir = bullet_info->dir;

    // convert to tile

    short init_inc = 0b010;
    short inc = 0b0010;

```

```

int result;
short offset = 0b100;

if (dir == 0b00) {

    cur_coords -= init_inc;
    cur_coords += offset * 256;
}
else if (dir == 0b01) {

    cur_coords += init_inc;
    cur_coords += offset * 256;
}
else if (dir == 0b10) {

    cur_coords -= init_inc * 256;
    cur_coords += offset;
}
else {

    cur_coords += init_inc * 256;
    cur_coords += offset;
}

for (;;) {

    if (dir == 0b00) {

        if (check_collision_wall(cur_coords) == 1) {

            result = 0;
            break;
        }

        else if (check_collision_tank(cur_coords, coords2) == 1) {

            result = 1;
            break;
        }

        else {

            color.pl_bulletl = cur_coords;
            color.pl_bulletd = 0b1000000000000000;
            set_background_color(&color);
            usleep(17000);

            cur_coords -= inc;

        }

    }

    else if (dir == 0b01) {

```

```

    if (check_collision_wall(cur_coords) == 1) {

        result = 0;
        break;
    }

    else if (check_collision_tank(cur_coords, coords2) == 1) {

        result = 1;
        break;
    }

    else {

        color.pl_bullet1 = cur_coords;
        color.pl_bulletd = 0b1000000000000000;
        set_background_color(&color);
        usleep(17000);

        cur_coords += inc;
    }

}

else if (dir == 0b10) {

    if (check_collision_wall(cur_coords) == 1) {

        result = 0;
        break;
    }

    else if (check_collision_tank(cur_coords, coords2) == 1) {

        result = 1;
        break;
    }

    else {

        color.pl_bullet1 = cur_coords;
        color.pl_bulletd = 0b1000000000000000;
        set_background_color(&color);
        usleep(17000);

        cur_coords -= inc * 256;
    }

}

else {

    if (check_collision_wall(cur_coords) == 1) {

        result = 0;
        break;
    }

```

```

    }

    else if (check_collision_tank(cur_coords, coords2) == 1) {

        result = 1;
        break;
    }

    else {

        color.p1_bulletl = cur_coords;
        color.p1_bulletd = 0b1000000000000000;
        set_background_color(&color);
        usleep(17000);

        cur_coords += inc * 256;
    }

}

//color.p1_bulletl = 0b0000000000000000;
color.p1_bulletd = 0b0000000000000000;
set_background_color(&color);

p2_hit = result;
p1_has_fired = 0;
}

void *fire_bullet_p2(void *args) {

    p1_hit = 0;

    struct bullet_args *bullet_info = (struct bullet_args *) args;
    unsigned short cur_coords = bullet_info->cur_coords;
    int dir = bullet_info->dir;

    // convert to tile

    short init_inc = 0b010;
    short inc = 0b0010;
    int result;
    short offset = 0b100;

    if (dir == 0b00) {

        cur_coords -= init_inc;
        cur_coords += offset * 256;
    }
    else if (dir == 0b01) {

        cur_coords += init_inc;
        cur_coords += offset * 256;
    }
}

```



```

else if (dir == 0b10) {
    cur_coords -= init_inc * 256;
    cur_coords += offset;
}
else {
    cur_coords += init_inc * 256;
    cur_coords += offset;
}

for (;;) {
    if (dir == 0b00) {
        if (check_collision_wall(cur_coords) == 1) {
            result = 0;
            break;
        }

        else if (check_collision_tank(cur_coords, coords) == 1) {
            result = 1;
            break;
        }

        else {
            color.p2_bullet1 = cur_coords;
            color.p2_bulletd = 0b1000000000000000;
            set_background_color(&color);
            usleep(17000);

            cur_coords -= inc;
        }

    }
    else if (dir == 0b01) {
        if (check_collision_wall(cur_coords) == 1) {
            result = 0;
            break;
        }

        else if (check_collision_tank(cur_coords, coords) == 1) {
            result = 1;
            break;
        }

        else {

```

```

        color.p2_bullet1 = cur_coords;
        color.p2_bulletd = 0b1000000000000000;
        set_background_color(&color);
        usleep(17000);

        cur_coords += inc;
    }

}
else if (dir == 0b10) {

    if (check_collision_wall(cur_coords) == 1) {

        result = 0;
        break;
    }

    else if (check_collision_tank(cur_coords, coords) == 1) {

        result = 1;
        break;
    }

    else {

        color.p2_bullet1 = cur_coords;
        color.p2_bulletd = 0b1000000000000000;
        set_background_color(&color);
        usleep(17000);

        cur_coords -= inc * 256;
    }

}
else {

    if (check_collision_wall(cur_coords) == 1) {

        result = 0;
        break;
    }

    else if (check_collision_tank(cur_coords, coords) == 1) {

        result = 1;
        break;
    }

    else {

        color.p2_bullet1 = cur_coords;
        color.p2_bulletd = 0b1000000000000000;
        set_background_color(&color);
        usleep(17000);
    }
}

```

```

    cur_coords += inc * 256;
    }

    }
}

//color.p2_bulletl = 0b0000000000000000;
color.p2_bulletd = 0b0000000000000000;
set_background_color(&color);

p2_has_fired = 0;
p1_hit = result;
}

int main()
{
    vga_ball_arg_t vla;
    static const char filename[] = "/dev/vga_ball";

    static const vga_ball_color_t colors[] = {
        { 0xff, 0x00, 0x00 }, /* Red */
        { 0x00, 0xff, 0x00 }, /* Green */
        { 0x00, 0x00, 0xff }, /* Blue */
        { 0xff, 0xff, 0x00 }, /* Yellow */
        { 0x00, 0xff, 0xff }, /* Cyan */
        { 0xff, 0x00, 0xff }, /* Magenta */
        { 0x80, 0x80, 0x80 }, /* Gray */
        { 0x00, 0x00, 0x00 }, /* Black */
        { 0xff, 0xff, 0xff } /* White */
    };

    # define COLORS 9

    printf("Tanks Game Userspace program started\n");

    if ( (vga_ball_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }

    struct controller_list devices = open_controllers();

    struct controller_pkt pkt1, pkt2;
    int fields1, fields2;
    int size1 = sizeof(pkt1);
    int size2 = sizeof(pkt2);

    for (;;) {

map_num = 0;

```

```

unsigned char h_coords;
unsigned char v_coords;

short inc = 0b1000;

int p1_left = 0;
int p1_right = 0;
int p1_up = 0;
int p1_down = 0;
int p1_fire = 0;
int p2_left = 0;
int p2_right = 0;
int p2_up = 0;
int p2_down = 0;
int p2_fire = 0;

int p1_score = 0;
int p2_score = 0;

int time = 80000;

int starting_press = 12;

coords = 0b0010000000100000;
unsigned short data = 0b1100000000000000;
coords2 = 0b0111100000100000;
unsigned short data2 = 0b1000000000000000;
unsigned short wall = 0b0000000000000000;
unsigned short misc = 0b0000000000000000;

color.wall = wall;
color.misc = misc;
color.p1_tankl = coords;
color.p1_tankd = data;
color.p2_tankl = coords2;
color.p2_tankd = data2;
color.p1_bulletl = 0;
color.p1_bulleted = 0;
color.p2_bulletl = 0;
color.p2_bulleted = 0;

set_background_color(&color);

// pre-game loop

for (;;) {

    //map_num = 0;

    libusb_interrupt_transfer(devices.device1, devices.device1_addr,
(unsigned char *) &pkt1, size1, &fields1, 0);
    libusb_interrupt_transfer(devices.device2, devices.device2_addr,
(unsigned char *) &pkt2, size2, &fields2, 0);

```

```

if (fields1 == 7 && fields2 == 7) {

uint8_t a = pkt1.xyab;

if (pkt1.v_arrows == 0) {

p1_up += 1;
if (p1_up >= 12) {
p1_up = 0;

if (wall == 0b0000000000000000) {

map_num = 1;
wall = 0b0100000000000000;
time = 50000;
}
else if (wall == 0b0100000000000000) {

map_num = 2;
wall = 0b1000000000000000;
time = 30000;
}

color.wall = wall;
set_background_color(&color);

coords = 0b0010000000100000;
data = 0b1100000000000000;
coords2 = 0b0111100000100000;
data2 = 0b1000000000000000;

while (coords <= 0b0100100000100000) {

color.p1_tankl = coords;
color.p1_tankd = data;
color.p2_tankl = coords2;
color.p2_tankd = data2;
set_background_color(&color);
usleep(time);

coords += inc * 256;
coords2 -= inc * 256;
}

data = 0b0100000000000000;
data2 = 0b0100000000000000;

while (coords <= 0b0100100001001000) {

color.p1_tankl = coords;
color.p1_tankd = data;
color.p2_tankl = coords2;
color.p2_tankd = data2;
set_background_color(&color);
}
}

```

```

        usleep(time);

        coords += inc;
        coords2 += inc;
    }

    data = 0b1000000000000000;
    data2 = 0b1100000000000000;

    while (coords >= 0b0010000001001000) {

        color.p1_tankl = coords;
        color.p1_tankd = data;
        color.p2_tankl = coords2;
        color.p2_tankd = data2;
        set_background_color(&color);
        usleep(time);

        coords -= inc * 256;
        coords2 += inc * 256;
    }

    data = 0b0000000000000000;
    data2 = 0b0000000000000000;

    while (coords >= 0b0010000000100000) {

        color.p1_tankl = coords;
        color.p1_tankd = data;
        color.p2_tankl = coords2;
        color.p2_tankd = data2;
        set_background_color(&color);
        usleep(time);

        coords -= inc;
        coords2 -= inc;
    }

}

}

} else if (pkt1.v_arrows == 255) {

    p1_down += 1;
    if (p1_down >= 12) {
        p1_down = 0;

        if (wall == 0b1000000000000000) {

            map_num = 1;
            wall = 0b0100000000000000;
            time = 50000;
        }
        else if (wall == 0b0100000000000000) {

```

```

    map_num = 0;
    wall = 0b0000000000000000;
    time = 80000;
}

color.wall = wall;
set_background_color(&color);

coords = 0b0010000000100000;
data = 0b1100000000000000;
coords2 = 0b0111100000100000;
data2 = 0b1000000000000000;

while (coords <= 0b0100100000100000) {

    color.p1_tankl = coords;
    color.p1_tankd = data;
    color.p2_tankl = coords2;
    color.p2_tankd = data2;
    set_background_color(&color);
    usleep(time);

    coords += inc * 256;
    coords2 -= inc * 256;
}

data = 0b0100000000000000;
data2 = 0b0100000000000000;

while (coords <= 0b01001000001001000) {

    color.p1_tankl = coords;
    color.p1_tankd = data;
    color.p2_tankl = coords2;
    color.p2_tankd = data2;
    set_background_color(&color);
    usleep(time);

    coords += inc;
    coords2 += inc;
}

data = 0b1000000000000000;
data2 = 0b1100000000000000;

while (coords >= 0b0010000001001000) {

    color.p1_tankl = coords;
    color.p1_tankd = data;
    color.p2_tankl = coords2;
    color.p2_tankd = data2;
    set_background_color(&color);
    usleep(time);
}

```

```

        coords -= inc * 256;
        coords2 += inc * 256;
    }

    data = 0b0000000000000000;
    data2 = 0b0000000000000000;

    while (coords >= 0b0010000000100000) {

        color.p1_tankl = coords;
        color.p1_tankd = data;
        color.p2_tankl = coords2;
        color.p2_tankd = data2;
        set_background_color(&color);
        usleep(time);

        coords -= inc;
        coords2 -= inc;
    }

}

} else if (a == 47 || a == 63 || a == 111 || a == 127 || a == 175 || a
== 191 || a == 239 || a == 255) {

    p1_fire += 1;
    if (p1_fire >= 15) {
        p1_fire = 0;

printf("Selected Map #%d...\n", map_num + 1);

        if (map_num == 0) {

            misc = 0b0000000000000000;
            wall = 0b0000000000000100;
        }
        else if (map_num == 1) {

            misc = 0b0000000000000001;
            wall = 0b0100000000000100;
        }
        else {

            misc = 0b0000000000000010;
            wall = 0b1000000000000100;
        }

        break;
    }

}
}

```



```

    }
}

coords = 0b0000100001101000;
data = 0b0000000000000000;
coords2 = 0b1001000000010000;
data2 = 0b0100000000000000;

color.wall = wall;
color.misc = misc;
color.p1_tankl = coords;
color.p1_tankd = data;
color.p2_tankl = coords2;
color.p2_tankd = data2;

set_background_color(&color);

for (;;) {
    if (p1_hit == 1) {
        p2_score += 1;

        unsigned short h = coords >> 9;
        unsigned char v = coords;

        unsigned char v_trunc = coords >> 1;

        unsigned int h_int = h * 512;
        unsigned int v_int = v_trunc * 4;

        unsigned short ex_coords = h_int + v_int + misc;

        color.misc = ex_coords;
        wall |= 0b0000000000010000;
        color.wall = wall;
        set_background_color(&color);

        usleep(150000);

        wall |= 0b0000000001000000;
        color.wall = wall;
        set_background_color(&color);

        usleep(150000);

        wall |= 0b0000000010000000;
        wall &= 0b1111111110111111;
        color.wall = wall;
        set_background_color(&color);
    }
}

```

```

// join threads so that hit gets reset

pthread_cancel(p1b);
p1_hit = 0;
p2_hit = 0;

color.p1_bulletd = 0b0000000000000000;
set_background_color(&color);
p1_has_fired = 0;

usleep(150000);

if (p2_score == 5) {

    wall |= 0b0000010000100000;
    wall &= 0b1111110011111111;
    color.wall = wall;
    set_background_color(&color);

    p1_score = 0;
    p2_score = 0;

    usleep(250000);

    // reset wall for next time

    printf("P2 wins!\n");

break;

}
else if (p2_score == 4) {

    wall |= 0b0000001100000000;
    color.wall = wall;
    set_background_color(&color);

    usleep(300000);
}

else if (p2_score == 3) {

    wall |= 0b0000001000000000;
    wall &= 0b1111111011111111;
    color.wall = wall;
    set_background_color(&color);

    usleep(300000);
}

else if (p2_score == 2) {

    wall |= 0b0000000100000000;
    color.wall = wall;

```

```

    set_background_color(&color);

    usleep(300000);
}

else if (p2_score == 1) {

    wall |= 0b0000000000000010;
    color.wall = wall;
    set_background_color(&color);

    usleep(300000);
}

// reset back to starting positions

wall &= 0b1111111100101111;
coords = 0b0000100001101000;
data = 0b0000000000000000;
coords2 = 0b1001000000010000;
data2 = 0b0100000000000000;
color.wall = wall;
color.p1_tankl = coords;
color.p1_tankd = data;
color.p2_tankl = coords2;
color.p2_tankd = data2;
set_background_color(&color);
}

else if (p2_hit == 1) {

    p1_score += 1;

    unsigned short h = coords2 >> 9;
    unsigned char v = coords2;

    unsigned char v_trunc = coords2 >> 1;

    unsigned int h_int = h * 512;
    unsigned int v_int = v_trunc * 4;

    unsigned short ex_coords = h_int + v_int + misc;

    color.misc = ex_coords;

    wall |= 0b0000000000010000;
    color.wall = wall;
    set_background_color(&color);

    usleep(150000);

    wall |= 0b0000000001000000;
    color.wall = wall;

```

```

set_background_color(&color);

usleep(150000);

wall |= 0b0000000010000000;
wall &= 0b1111111101111111;
color.wall = wall;
set_background_color(&color);

pthread_cancel(p2b);
p1_hit = 0;
p2_hit = 0;

color.p2_bulletd = 0b0000000000000000;
set_background_color(&color);
p2_has_fired = 0;

usleep(150000);

if (p1_score == 5) {

    wall |= 0b0010000000100000;
    wall &= 0b1110011111111111;
    color.wall = wall;
    set_background_color(&color);

    p1_score = 0;
    p2_score = 0;

    usleep(250000);

    // reset wall for next time

printf("P1 wins!\n");

    break;

}
else if (p1_score == 4) {

    wall |= 0b0001100000000000;
    color.wall = wall;
    set_background_color(&color);

    usleep(300000);
}

else if (p1_score == 3) {

    wall |= 0b0001000000000000;
    wall &= 0b1111011111111111;
    color.wall = wall;
    set_background_color(&color);

```

```

    usleep(300000);
}

else if (p1_score == 2) {

    wall |= 0b0000100000000000;
    color.wall = wall;
    set_background_color(&color);

    usleep(300000);
}

else if (p1_score == 1) {

    wall |= 0b00000000000001000;
    color.wall = wall;
    set_background_color(&color);

    usleep(300000);
}

// reset back to starting positions

wall &= 0b1111111100101111;
coords = 0b0000100001101000;
data = 0b0000000000000000;
coords2 = 0b1001000000010000;
data2 = 0b0100000000000000;
color.wall = wall;
color.p1_tankl = coords;
color.p1_tankd = data;
color.p2_tankl = coords2;
color.p2_tankd = data2;
set_background_color(&color);
}

libusb_interrupt_transfer(devices.device1, devices.device1_addr,
(unsigned char *) &pkt1, size1, &fields1, 0);
libusb_interrupt_transfer(devices.device2, devices.device2_addr,
(unsigned char *) &pkt2, size2, &fields2, 0);

if (fields1 == 7 && fields2 == 7) {

    // do detecting pkts for each

    uint8_t a = pkt1.xyab;

    // Check left/right arrows (can only be one at a time)
    if (pkt1.h_arrows == 0) {

        p1_left += 1;
        if (p1_left >= 12) {
            p1_left = 0;
        }
    }
}

```

```

        // left
h_coords = coords >> 8;

        if (h_coords > 0b00000000) {
            coords -= inc * 256;

            if (check_collision_wall(coords) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data = 0b1000000000000000;

                color.pl_tankl = coords;
                color.pl_tankd = data;
                set_background_color(&color);
            }
            else {

                coords += inc * 256;
            }
        }
    }

} else if (pkt1.h_arrows == 255) {

    pl_right += 1;
    if (pl_right >= 12) {
        pl_right = 0;

        // right
h_coords = coords >> 8;

        if (h_coords < 0b10011000) {

            coords += inc * 256;

            if (check_collision_wall(coords) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data = 0b1100000000000000;

                color.pl_tankl = coords;
                color.pl_tankd = data;
                set_background_color(&color);
            }
            else {

                coords -= inc * 256;
            }
        }
    }
}

```

```

    }
}

// Check up/down arrows (can only be one at a time)
else if (pkt1.v_arrows == 0) {

    pl_up += 1;
    if (pl_up >= 12) {
        pl_up = 0;

        // up

        v_coords = coords;

        if (v_coords > 0b00000000) {

            coords -= inc;

            if (check_collision_wall(coords) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data = 0b0000000000000000;

                color.pl_tankl = coords;
                color.pl_tankd = data;
                set_background_color(&color);
            }
            else {

                coords += inc;
            }
        }
    }

} else if (pkt1.v_arrows == 255) {

    pl_down += 1;
    if (pl_down >= 12) {
        pl_down = 0;

        // down

        v_coords = coords;

        if (v_coords < 0b01110000) {

            coords += inc;

            if (check_collision_wall(coords) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data = 0b0100000000000000;

```

```

        color.pl_tankl = coords;
        color.pl_tankd = data;
        set_background_color(&color);
    }
    else {

        coords -= inc;
    }
}
}

// Check if shoot button (A) is pressed
else if (a == 47 || a == 63 || a == 111 || a == 127 || a == 175 ||
a == 191 || a == 239 || a == 255) {

    p1_fire += 1;
    if (p1_fire >= 15) {
        p1_fire = 0;
    }

    if (p1_has_fired == 0) {

        p1_has_fired = 1;

        args1.cur_coords = coords;
        args1.dir = data >> 14;

        pthread_create(&p1b, NULL, &fire_bullet_p1, (void *)
&args1);
    }
}

}

uint8_t b = pkt2.xyab;

// Check left/right arrows (can only be one at a time)
if (pkt2.h_arrows == 0) {

    p2_left += 1;
    if (p2_left >= 12) {
        p2_left = 0;
    }

    // left

    h_coords = coords2 >> 8;

    if (h_coords > 0b00000000) {

```



```

        coords2 -= inc * 256;

        if (check_collision_wall(coords2) == 0 &&
check_collision_tank(coords, coords2) == 0) {

            data2 = 0b1000000000000000;

            color.p2_tankl = coords2;
            color.p2_tankd = data2;
            set_background_color(&color);
        }
        else {

            coords2 += inc * 256;
        }
    }
}

} else if (pkt2.h_arrows == 255) {

    p2_right += 1;
    if (p2_right >= 12) {
        p2_right = 0;

        // right

        h_coords = coords2 >> 8;

        if (h_coords < 0b10011000) {

            coords2 += inc * 256;

            if (check_collision_wall(coords2) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data2 = 0b1100000000000000;

                color.p2_tankl = coords2;
                color.p2_tankd = data2;
                set_background_color(&color);
            }
            else {

                coords2 -= inc * 256;
            }
        }
    }
}

}

// Check up/down arrows (can only be one at a time)
else if (pkt2.v_arrows == 0) {

```

```

    p2_up += 1;
    if (p2_up >= 12) {
        p2_up = 0;

        // up

v_coords = coords2;

        if (v_coords > 0b00000000) {

            coords2 -= inc;

            if (check_collision_wall(coords2) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data2 = 0b0000000000000000;

                color.p2_tankl = coords2;
                color.p2_tankd = data2;
                set_background_color(&color);
            }
            else {

                coords2 += inc;
            }
        }
    }

} else if (pkt2.v_arrows == 255) {

    p2_down += 1;
    if (p2_down >= 12) {
        p2_down = 0;

        // down

v_coords = coords2;

        if (v_coords < 0b01110000) {

            coords2 += inc;

            if (check_collision_wall(coords2) == 0 &&
check_collision_tank(coords, coords2) == 0) {

                data2 = 0b0100000000000000;

                color.p2_tankl = coords2;
                color.p2_tankd = data2;
                set_background_color(&color);
            }
            else {

                coords2 -= inc;

```

```

        }
    }
}

    // Check if shoot button (A) is pressed
    else if (b == 47 || b == 63 || b == 111 || b == 127 || b == 175 ||
b == 191 || b == 239 || b == 255) {

        p2_fire += 1;
        if (p2_fire >= 15) {
            p2_fire = 0;

            if (p2_has_fired == 0) {

                p2_has_fired = 1;

                args2.cur_coords = coords2;
                args2.dir = data2 >> 14;

                pthread_create(&p2b, NULL, &fire_bullet_p2, (void *)
&args2);

            }
        }

    }

}
}

printf("Tanks Game Userspace program terminating\n");
return 0;

}

```

## 5. controller.h

```

#ifndef _CONTROLLER_H
#define _CONTROLLER_H

#include <stdio.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h>

```

```

struct controller_list {

    struct libusb_device_handle *device1;
    struct libusb_device_handle *device2;
    uint8_t device1_addr;
    uint8_t device2_addr;

};

struct controller_pkt {

    uint8_t const1;
    uint8_t const2;
    uint8_t const3;
    uint8_t h_arrows;
    uint8_t v_arrows;
    uint8_t xyab;
    uint8_t rl;

};

struct args_list {

    struct controller_list devices;
    char * buttons;
    int mode;
    int print;

};

extern struct controller_list open_controller(uint8_t *);

#endif

```

## 6. controller.c

```

#include "controller.h"
#include <libusb-1.0/libusb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct controller_list open_controllers() {

    printf("Searching for USB connections...\n");

    uint8_t endpoint_address = 0;
    struct controller_list devices;
    libusb_device **devs;
    struct libusb_device_descriptor desc;
    struct libusb_device_handle *controller = NULL;
    ssize_t num_devs;

```

```

// Boot libusb library
if (libusb_init(NULL) != 0) {
    printf("\nERROR: libusb failed to boot");
    exit(1);
}

if ((num_devs = libusb_get_device_list(NULL, &devs)) < 0) {
    printf("\nERROR: no controllers found");
    exit(1);
}

//printf("Detected %d devices...\n", num_devs);
int connection_count = 0;
for (int i = 0; i < num_devs; i++) {

    libusb_device *dev = devs[i];

    if (libusb_get_device_descriptor(dev, &desc) < 0) {
        printf("\nERROR: bad device descriptor.");
        exit(1);
    }

    // Our controllers have idProduct of 17
    if (desc.idProduct == 17) {

        //printf("FOUND: idProduct-%d ", desc.idProduct);
        struct libusb_config_descriptor *config;
        if ((libusb_get_config_descriptor(dev, 0,
&config)) < 0) {
            printf("\nERROR: bad config descriptor.");
            exit(1);
        }
        //printf("interfaces-%d\n",
config->bNumInterfaces);

        // Our controllers only have a single interface,
no need for looping
        // This interface also only has one
.num_altsetting, no need for looping

        int r;
        const struct libusb_interface_descriptor *inter =
config->interface[0].altsetting;
        if ((r = libusb_open(dev, &controller)) != 0) {
            printf("\nERROR: couldn't open
controller");
            exit(1);
        }
        if (libusb_kernel_driver_active(controller, 0)) {
            libusb_detach_kernel_driver(controller,
0);
        }
    }
}

```

```

        libusb_set_auto_detach_kernel_driver(controller,
0);
        if ((r = libusb_claim_interface(controller, 0)) !=
0) {
            printf("\nERROR: couldn't claim
controller.");
            exit(1);
        }

        endpoint_address =
inter->endpoint[0].bEndpointAddress;
        connection_count++;

        if (connection_count == 1) {
            devices.device1 = controller;
            devices.device1_addr = endpoint_address;
        } else {
            devices.device2 = controller;
            devices.device2_addr = endpoint_address;

//printf("%d:%d,%d:%d\n", devices.device1, devices.device1_addr, devices.device2, devices.device2_addr)
            goto found;
        }
    }

    if (connection_count < 2) {
        printf("ERROR: couldn't find 2 controllers.");
        exit(1);
    }

    found:
        printf("Connected %d controllers!\n", connection_count);
        libusb_free_device_list(devs, 1);

    return devices;
}

void detect_presses(struct controller_pkt pkt, char *buttons, int mode) {

    // Choose whether you want human-readable or binary output
    char vals[] = "LRUDA";
    if (mode == 1) {
        strcpy(buttons, "00000");
        strcpy(vals, "11111");
    } else {
        strcpy(buttons, "_____");
    }

    // Check left/right arrows (can only be one at a time)
    if (pkt.h_arrows == 0) {
        buttons[0] = vals[0];
    } else if (pkt.h_arrows == 255) {
        buttons[1] = vals[1];
    }
}

```

```

    }

    // Check up/down arrows (can only be one at a time)
    if (pkt.v_arrows == 0) {
        buttons[2] = vals[2];
    } else if (pkt.v_arrows == 255) {
        buttons[3] = vals[3];
    }

    // Check if shoot button (A) is pressed
    uint8_t a = pkt.xyab;
    if (a == 47 || a == 63 || a == 111 || a == 127 || a == 175 || a ==
191 || a == 239 || a == 255) {
        buttons[4] = vals[4];
    }

    //printf("\n%s", buttons);
}

void *listen_controllers(void *arg) {

    struct args_list *args_p = arg;
    struct args_list args = *args_p;
    struct controller_list devices = args.devices;

    struct controller_pkt pkt1, pkt2;
    int fields1, fields2;
    int size1 = sizeof(pkt1);
    int size2 = sizeof(pkt2);
    char buttons1[] = "_____";
    char buttons2[] = "_____";

    for (;;) {

        libusb_interrupt_transfer(devices.device1,
devices.device1_addr, (unsigned char *) &pkt1, size1, &fields1, 0);
        libusb_interrupt_transfer(devices.device2,
devices.device2_addr, (unsigned char *) &pkt2, size2, &fields2, 0);

        // 7 fields should be transferred for each packet
        if (fields1 == 7 && fields2 == 7) {
            detect_presses(pkt1, buttons1, args.mode);
            detect_presses(pkt2, buttons2, args.mode);
            strcat(buttons1, buttons2);
            strcpy(args.buttons, buttons1);
            if (args.print == 1) {
                printf("%s\n", args.buttons);
            }
        }
    }
}

```