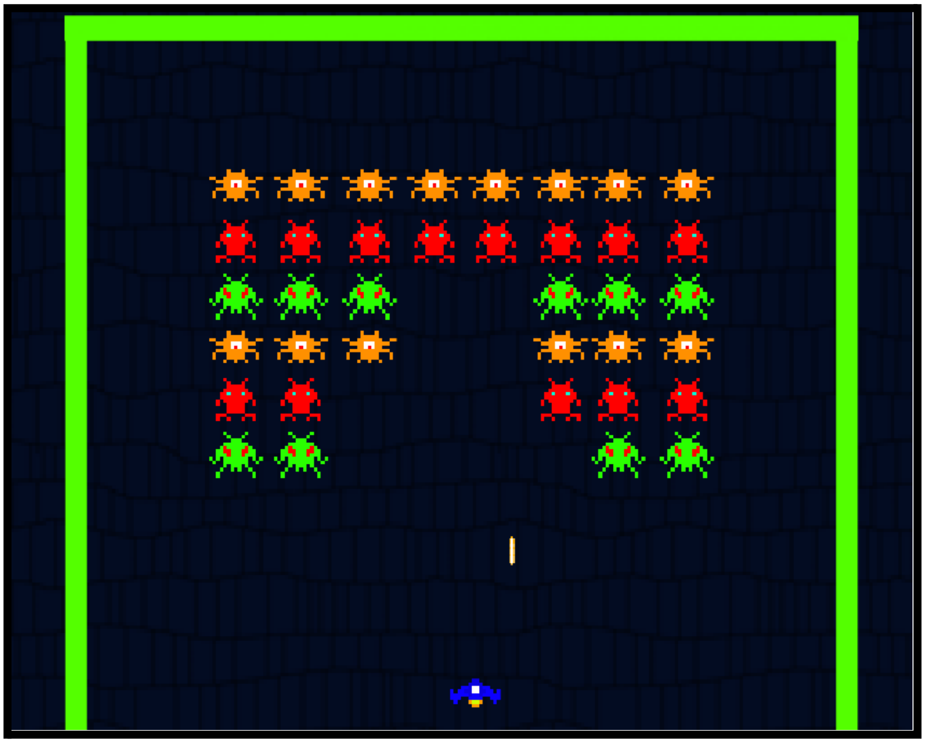
Space Invaders Revamp
Project Report



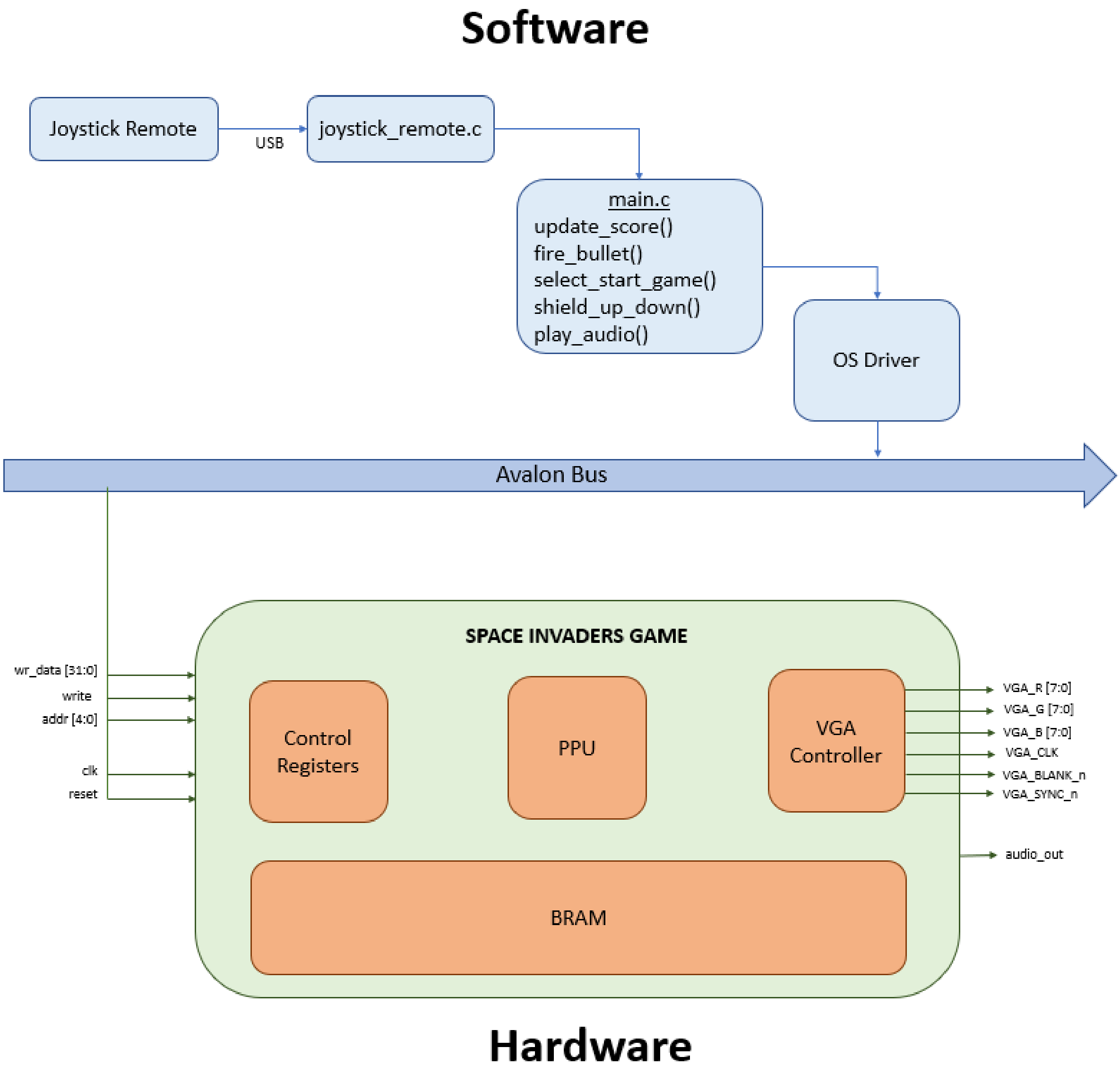
Alan Hwang (awh2135)
Zach Burpee (zcb2110)
Mili Sehgal (ms6557)

Project Overview

In this project, we have recreated the classic “Space Invaders” arcade game from the 1970s on the De1-SoC FPGA. In this game, the player controls a defender spaceship that moves horizontally across the bottom of the screen and fires missiles at enemy alien ships. Every few seconds, the enemy alien ships shift down and approach the defender spaceship. Additionally, the enemy alien ships also drop bombs that the defender must avoid when moving around at the bottom of the screen. To proceed to the next level, the player must destroy all of the enemy alien ships before they reach the bottom of the screen.

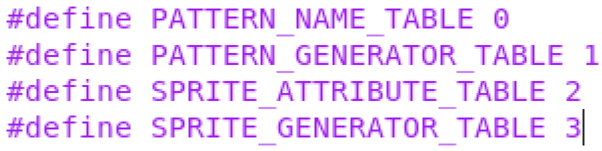
To recreate this game on FPGA, we used a retro NES USB controller to get the user input. To handle this, we wrote a device driver in software that correctly forwards the user input to the top level software logic. Our software controls all of the Space Invaders logic and passes the data to byte-addressable VRAM in hardware, which is then displayed on a VGA monitor.

Top-Level Architecture



Hardware

- Our approach for displaying the graphics data involved utilizing a tile-and-sprites method. This process is done with four tables: a pattern name, pattern generation, sprite attribute, and sprite generation table.



```
#define PATTERN_NAME_TABLE 0
#define PATTERN_GENERATOR_TABLE 1
#define SPRITE_ATTRIBUTE_TABLE 2
#define SPRITE_GENERATOR_TABLE 3
```

- Tiles were employed to display the user interface and gameplay messages. To accomplish this, we used a pattern generator table, which is addressed by 12 bits and results in 4096 rows. Since every pattern tile requires 32 bytes, we had the capacity to store up to 128 distinct patterns. A pattern name table parses the generator table and obtains the corresponding pattern attributes. The pattern table does not require all 12 bits of addressing; this was kept in case new patterns and UI features wanted to be added.

- The sprites are displayed in a similar structure to the pattern tiles, except we assumed that all of the sprites are moving components – unlike the tiles. Because Space Invaders has a lot of moving parts (20+ ships, missiles, bombs), we needed to create a very large sprite generator table. Each sprite requires 128 bytes and 32 address bits were required to create the rows in our sprite generator table. We used a sprite attribute table to store the addresses of each sprite. Each sprite attribute contains the y position, x position, and the sprite address from the generator table. A combinational block allows for colors to be prioritized and for sprites to be displayed in front of the tiles.

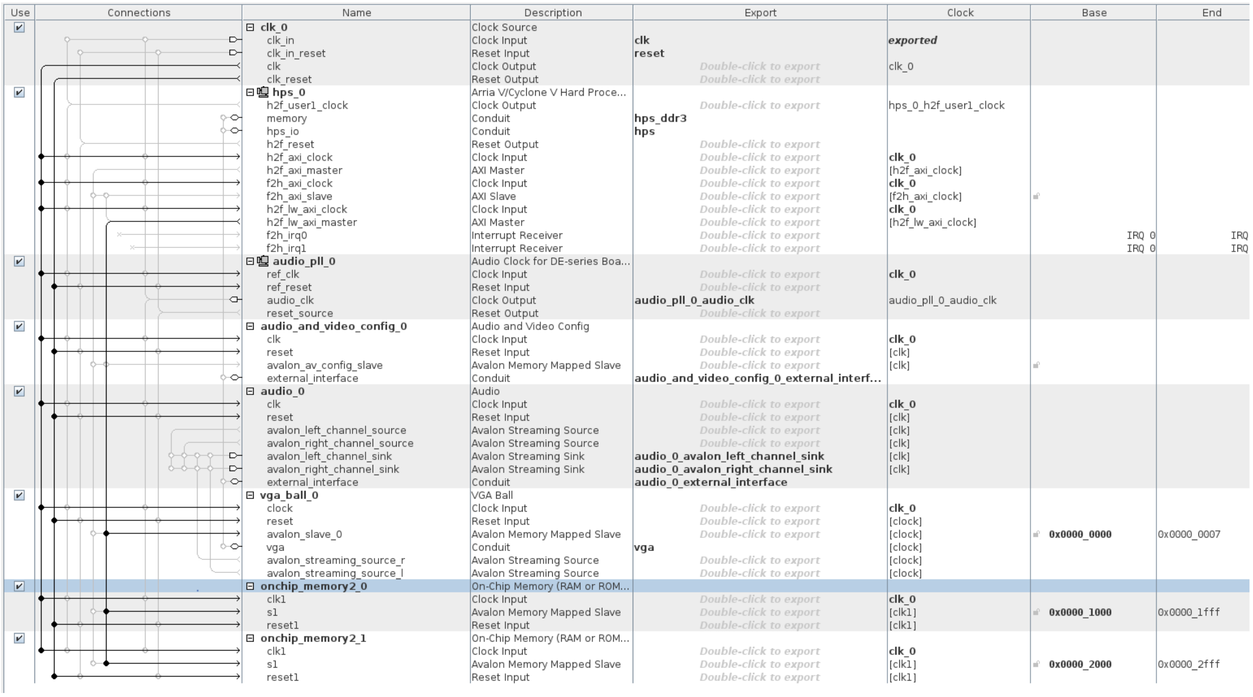
- Additionally, each sprite requires its own state machine. This is particularly tricky with Space Invaders, since there are many enemies and the horizontal count of a sprite must not overlap with another sprite. If a sprite needs to be displayed on the following line, the sprite generator table is accessed from the designated base address. The horizontal position of the sprite is then loaded into a down counter, while the sprite pixels are loaded into a shift register. When the next vertical line is reached, the down counter decreases, and a 4-bit pixel value is retrieved from the shift register, which corresponds to the 24-bit RGB color value. Since Space Invaders only needs green and white as colors, a color translation table was created, where a 4-bit pixel value maps to the 24-bit RGB value.

Avalon Bus: HW/SW Interface

- Our hardware interface accepts a 32 bit write packet from software that is structured as follows:
    1) Bits 0 - 1: Table Selector [pat_name, pat_gen, sprite_attr, sprite_gen]
    2) Bits 2 - 17: 16-bit Destination Address
    3) Bits 24 - 31: 8-bit Data to Write at Destination Address

AUDIO INTERFACE

Audio CODEC is interfaced with the Avalon bus using Avalon Stream Interface. Avalon-ST is an interface that supports the unidirectional flow of data, including multiplexed streams, packets, and DSP data. The audio streaming interface consists of 3 signals: data signal, read signal, and a valid signal. The data signal carries the actual audio data, while the valid signal indicates when the data is valid and should be processed, and the read signal is used to control the flow of the data.



Use	Connections	Name	Description	Export	Clock	Base	End
☑		⊟ **clk_0**	Clock Source				
		clk_in	Clock Input	**clk**	***exported***		
		clk_in_reset	Reset Input	**reset**			
		clk	Clock Output	*Double-click to export*	clk_0		
		clk_reset	Reset Output	*Double-click to export*			
☑		⊟ **hps_0**	Arria V/Cyclone V Hard Proce...				
		h2f_user1_clock	Clock Output	*Double-click to export*	hps_0_h2f_user1_clock		
		memory	Conduit	**hps_ddr3**			
		hps_io	Conduit	**hps**			
		h2f_reset	Reset Output	*Double-click to export*			
		h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_clock]		
		f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_clock]		
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi_clock]		
		f2h_irq0	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ
		f2h_irq1	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ
☑		⊟ **audio_pll_0**	Audio Clock for DE-series Boa...				
		ref_clk	Clock Input	*Double-click to export*	**clk_0**		
		ref_reset	Reset Input	*Double-click to export*			
		audio_clk	Clock Output	**audio_pll_0_audio_clk**	audio_pll_0_audio_clk		
		reset_source	Reset Output	*Double-click to export*			
☑		⊟ **audio_and_video_config_0**	Audio and Video Config				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		avalon_av_config_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]		
		external_interface	Conduit	**audio_and_video_config_0_external_interf...**			
☑		⊟ **audio_0**	Audio				
		clk	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clk]		
		avalon_left_channel_source	Avalon Streaming Source	*Double-click to export*	[clk]		
		avalon_right_channel_source	Avalon Streaming Source	*Double-click to export*	[clk]		
		avalon_left_channel_sink	Avalon Streaming Sink	**audio_0_avalon_left_channel_sink**	[clk]		
		avalon_right_channel_sink	Avalon Streaming Sink	**audio_0_avalon_right_channel_sink**	[clk]		
		external_interface	Conduit	**audio_0_external_interface**			
☑		⊟ **vga_ball_0**	VGA Ball				
		clock	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clock]		
		avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_0000**	0x0000_0007
		vga	Conduit	**vga**	[clock]		
		avalon_streaming_source_r	Avalon Streaming Source	*Double-click to export*	[clock]		
		avalon_streaming_source_l	Avalon Streaming Source	*Double-click to export*	[clock]		
☑		⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM...				
		clk1	Clock Input	*Double-click to export*	**clk_0**		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	**0x0000_1000**	0x0000_1fff
		reset1	Reset Input	*Double-click to export*	[clk1]		
☑		⊟ **onchip_memory2_1**	On-Chip Memory (RAM or ROM...				
		clk1	Clock Input	*Double-click to export*	**clk_0**		
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	**0x0000_2000**	0x0000_2fff
		reset1	Reset Input	*Double-click to export*	[clk1]		

Software

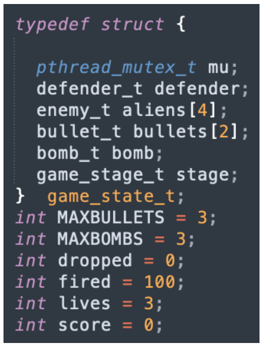
Game Logic

- Game Stages: (STAGE_MENU, STAGE_IN_GAME, STAGE_END)

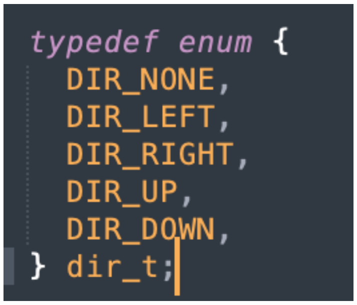
  The game was set to refresh every 8ms from a counter that iterated through the MAX_INT and modulo 40 division. The game interface keeps track of the current level, current lives, current points, and different audio tracks when an event occurs. The levels increase once all the enemy ships have been defeated. The lives will decrease after the player ship is hit with an enemy bomb. The points will increase according to the enemy ship defeated by the player ship. The audio tracks will be determined by an event taking place - for example, an enemy is damaged, the game is over, the player takes damage, and background music.

- Game State:

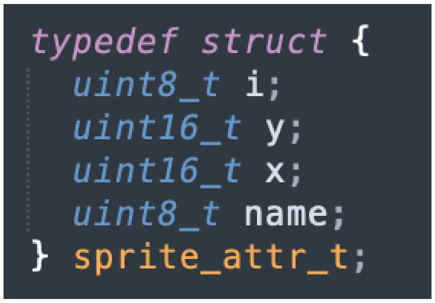
  The game state holds information related to the objects it must keep track of during updates and relevant inputs. The struct show below holds all the information:



```
typedef struct {

    pthread_mutex_t mu;
    defender_t defender;
    enemy_t aliens[4];
    bullet_t bullets[2];
    bomb_t bomb;
    game_stage_t stage;
}  game_state_t;
int MAXBULLETS = 3;
int MAXBOMBS = 3;
int dropped = 0;
int fired = 100;
int lives = 3;
int score = 0;
```



```
typedef enum {
    DIR_NONE,
    DIR_LEFT,
    DIR_RIGHT,
    DIR_UP,
    DIR_DOWN,
} dir_t;
```

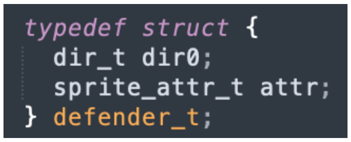


```
typedef struct {
    uint8_t i;
    uint16_t y;
    uint16_t x;
    uint8_t name;
} sprite_attr_t;
```

  The struct on the left shows the defender, the enemies, bullets, bombs, and game stage; along with settable parameters for bullets, bombs, lives, and score. The struct in the middle explains the direction attributes that are set within the different instances of the game state. The struct on the right explains the relevant attributes that sprite will take into consideration when creating the instances on screen.

- Defender Ship State & Function:

  The player ship will be “defending” the Earth by firing shots at the incoming enemy ships. The player ship receives directional commands and firing commands from the joystick peripheral in a struct. The player ship can be harmed by the incoming bullets from enemy ships and a life will be taken from the game interface. The player ship does not have a limited number of bullets, but it does have a cooldown on how fast the player can fire. Another counter has a cooldown period of 200ms. There is only one reference to an instance of the defender in the game state, so it is a non-array object with the attributes shown below.

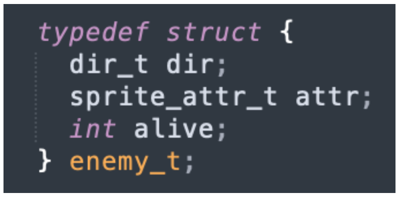


```
typedef struct {
  dir_t dir0;
  sprite_attr_t attr;
} defender_t;
```

As the defender ship is a pretty simple instance, it only requires sprite attributes and direction. The defender movement function will also call a check function each update cycle in order to evaluate if any bomb has hit the ship.

- Enemy Ship State & Function:

The enemy ships will be “invading” the Earth by slowly moving down toward the defending ship and dropping periodic bombs too. During the process, they “bounce” back and forth across the screen and turn directions each time the end ships hit the screen edge. Once the lowest ship reaches the player or if all the enemy ships are eliminated, the game is over. Additionally, different levels of enemy ships can appear as the player progresses, which will require more shots to defeat the enemy ship. The enemy ship state will have to maintain these hitpoint values. The bombs will be dropped with a 10% chance for each iteration, meaning 10 movements will result in a bomb being dropped. A maximum of 3 bombs can be on the screen at once and can be programmed accordingly. The defender is allowed to move at a constant rate of 3 pixels every 8ms.

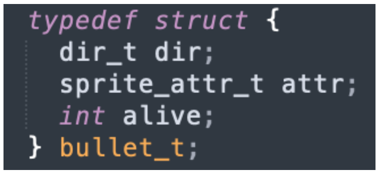


```
typedef struct {
  dir_t dir;
  sprite_attr_t attr;
  int alive;
} enemy_t;
```

The enemy instance is similar to the defender instance except for the lives being an attribute rather than an overall calculation. This is due to the fact that multiple instances of an enemy are produced and must be looped through during each update to check for events. The alive count will be increased as the levels get more difficult; when the alive count reaches 0, the enemy dies. Enemies are allowed to move at a starting rate of 2 pixels every 8ms.

- Bullet State & Function:

The bullet is instantiated once the controller button A has been pressed by the user. The function will check each bullet instance in the game state class and determine if a maximum number of bullets have already been called. If there is room for another bullet, the alive attribute is incremented on the bullet class to signal it has been instantiated. The bullet is fired directly from the cannon of the defender toward enemy ships and propagates at a rate of 8 pixels per 8ms.

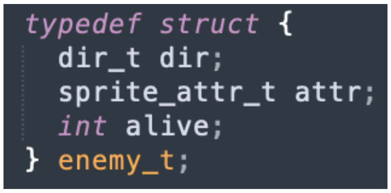


```c
typedef struct {
  dir_t dir;
  sprite_attr_t attr;
  int alive;
} bullet_t;
```

Similarly, the sprite and direction of the bullet are updated every time the bullet is re/instantiated during updates and relevant events. Once a bullet has hit an enemy or gone off screen, the alive attribute is decremented and the sprite disappears. It is then when a new bullet can be queued to be fired. There can only be a maximum of 3 bombs on the screen at once.

- Bomb State & Function:

  The bomb is instantiated once the probability that an enemy drops a bomb has been reached. The function will instantiate a bomb in the same place that the enemy is located. The bomb will propagate at a rate of 8 pixels per 8ms toward the defender.



```c
typedef struct {
  dir_t dir;
  sprite_attr_t attr;
  int alive;
} enemy_t;
```

  Following suit with the previous structs, the direction and sprite attributes are also listed here. Once a bomb has hit the defender or gone off screen, the bomb sprite is reset and the count is decremented.

- Setup Game & Reset Game Functions:

  The setup game and reset functions are one in the same once the game has been loaded in. The reset will iterate through all the class instances in the game state struct and assign appropriate starting attributes and locations to relevant characters. The sprite class uses a unique identifier that each unique ship/defender/bullet/bomb must be assigned before it is called for the first time. There is a series of for loops that assign these unique integers to each sprite struct. Each time the game is reset or started for the first time, this function is called and the ships are lined up on the top of the screen, with the defender on the bottom of the screen.

- Main State & Function:

  The main state function is a constant loop that updates relevant game states based on the current stage of the game (START, IN_GAME, END). For the start, the patterns are assigned to explain directions to start the game. Additionally, the screen is cleared from all previous sprites, the score is cleared, and the lives are reset back to 3. Once the START button is pressed on the controller, the game stage is set to IN_GAME, where the loop continually calls tracking functions for the defender, enemy, bullets, and bombs.

Additionally, the score and lives are constantly updated with every call. Once the game is ended (either by win or lose), the relevant integer indicating whether the game was a win or loss is passed into the END stage. The “win” screen will print out congratulations and the score. A “lose” screen will print out a game over and the score. Both end states will print out the instructions to reset the game. Once the reset button is clicked, the screen is cleared and the initial state of START is set and the while loop restarts at the beginning.

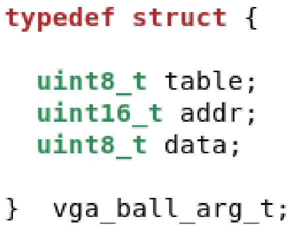
Gamepad Controller

The joystick peripheral must have communication algorithms that will relay important information for each button. The following functions will be implemented:

- move_left() → button movement will indicate left translation of pixels of player ship
- move_right() → button movement will indicate right translation of pixels of player ship
- fire_bullet() → button press will launch bullet pixels from player ship
- start() → start button will start game in beginning
- select() → select button will reset game after lives are terminated or game is won

Kernel Space Driver

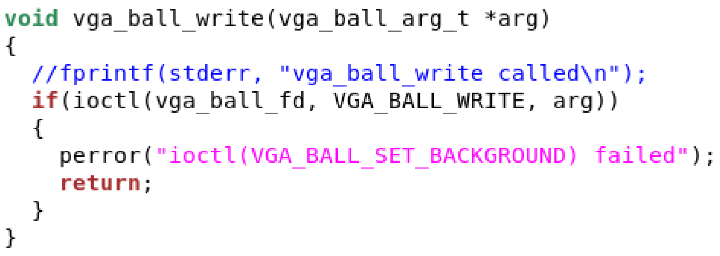
- The kernel driver follows the following struct: a uint8_t table, a uint16_t addr, and a uint8_t data field. These values are concatenated together to pass the 32 bits of write data to hardware.



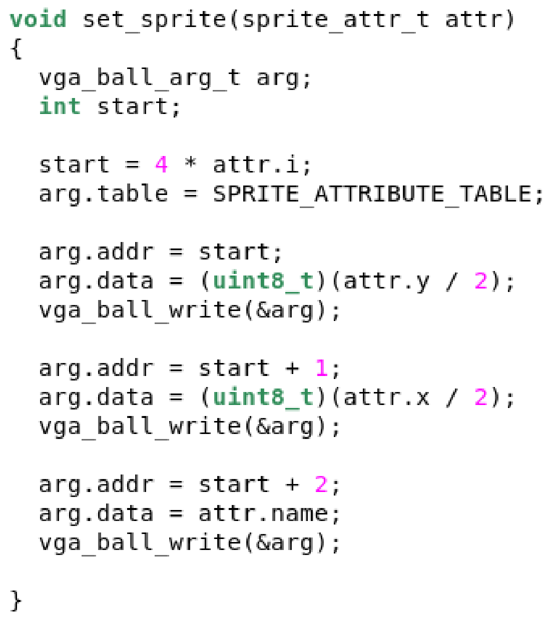
```
typedef struct {

  uint8_t table;
  uint16_t addr;
  uint8_t data;

}  vga_ball_arg_t;
```

User Space Driver

- In vga_ball_write, an ioctl call is made similar to the vga_ball done in Lab 3. This is called in set_sprite and set_pattern to pass the hardware the 32-bit packet containing the table, addr, and data information.



```
void vga_ball_write(vga_ball_arg_t *arg)
{
  //fprintf(stderr, "vga_ball_write called\n");
  if(ioctl(vga_ball_fd, VGA_BALL_WRITE, arg))
  {
    perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
    return;
  }
}
```



```
void set_sprite(sprite_attr_t attr)
{
  vga_ball_arg_t arg;
  int start;

  start = 4 * attr.i;
  arg.table = SPRITE_ATTRIBUTE_TABLE;

  arg.addr = start;
  arg.data = (uint8_t)(attr.y / 2);
  vga_ball_write(&arg);

  arg.addr = start + 1;
  arg.data = (uint8_t)(attr.x / 2);
  vga_ball_write(&arg);

  arg.addr = start + 2;
  arg.data = attr.name;
  vga_ball_write(&arg);

}
```

Lessons Learned

- Test hardware in parallel with other work! Compiling Quartus and copying the .dts and .rbf files to the FPGA is extremely time consuming. It's also very easy to lose track of what changes to hardware were made and why. When working on hardware, write out a set plan of implementation changes to try and keep track of the changes. While Quartus is compiling, work on software in parallel.

- Get the HW/SW interface working as soon as possible. Coding and understanding how software passes information to the hardware is vital to any video game project that requires a display. Following the interface, sprites and bitmapping can be implemented to see how changes in software display on the actual VGA peripheral.

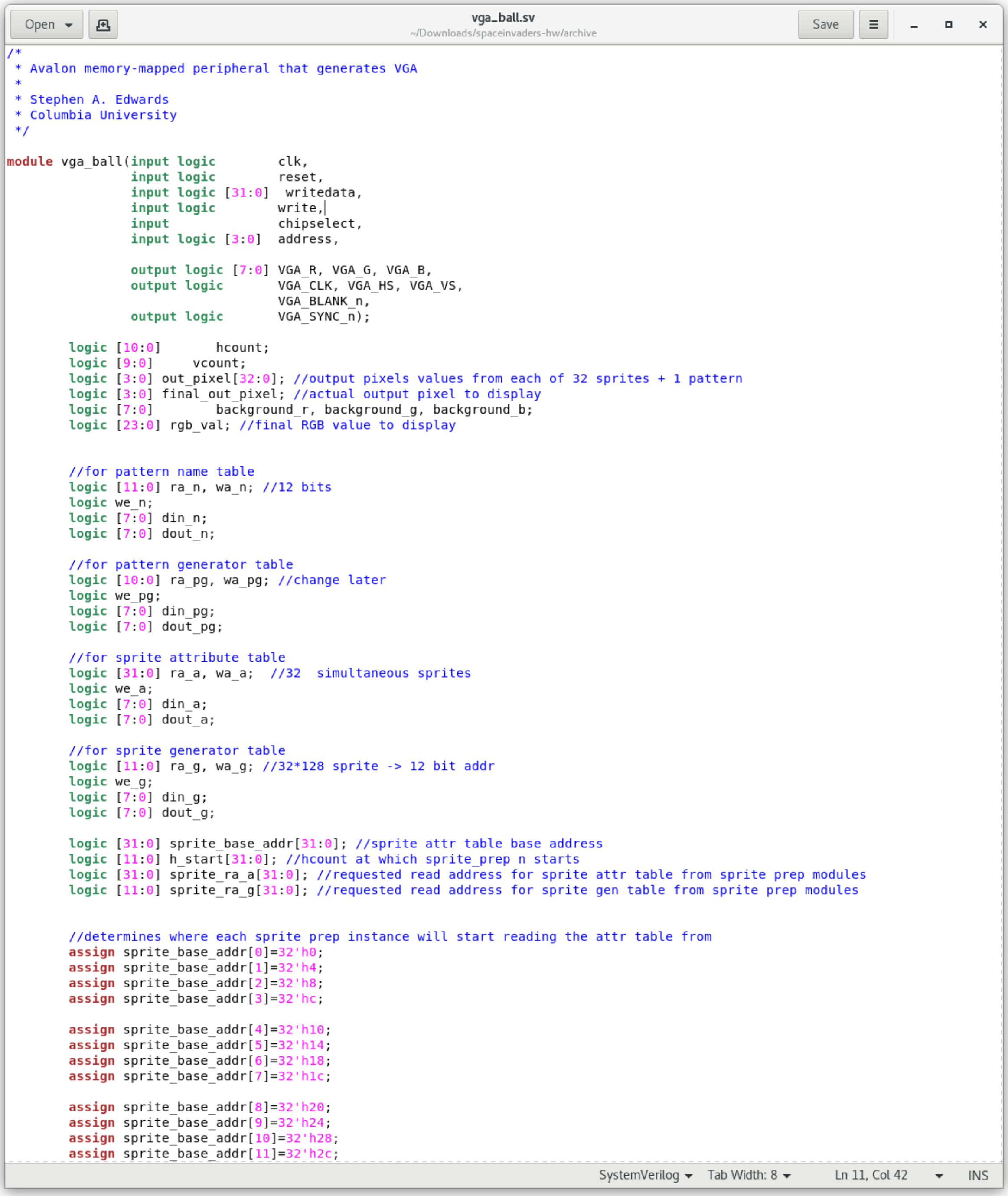
- Start with a strong basis on hardware. Once the hardware is correctly implemented with basic test cases in software, this will make the challenge of software testing an isolated experiment. During our programming, we progressed with the hardware at a level to comfortably test 5 sprites. We perfected the algorithms for 5 sprites assuming that adding more hardware to support more sprites would be intuitive. However, once the sprites would not perform as expected, the isolation of errors was now expanded to both the hardware and software. This made the process extremely time consuming and difficult.

Project Breakdown

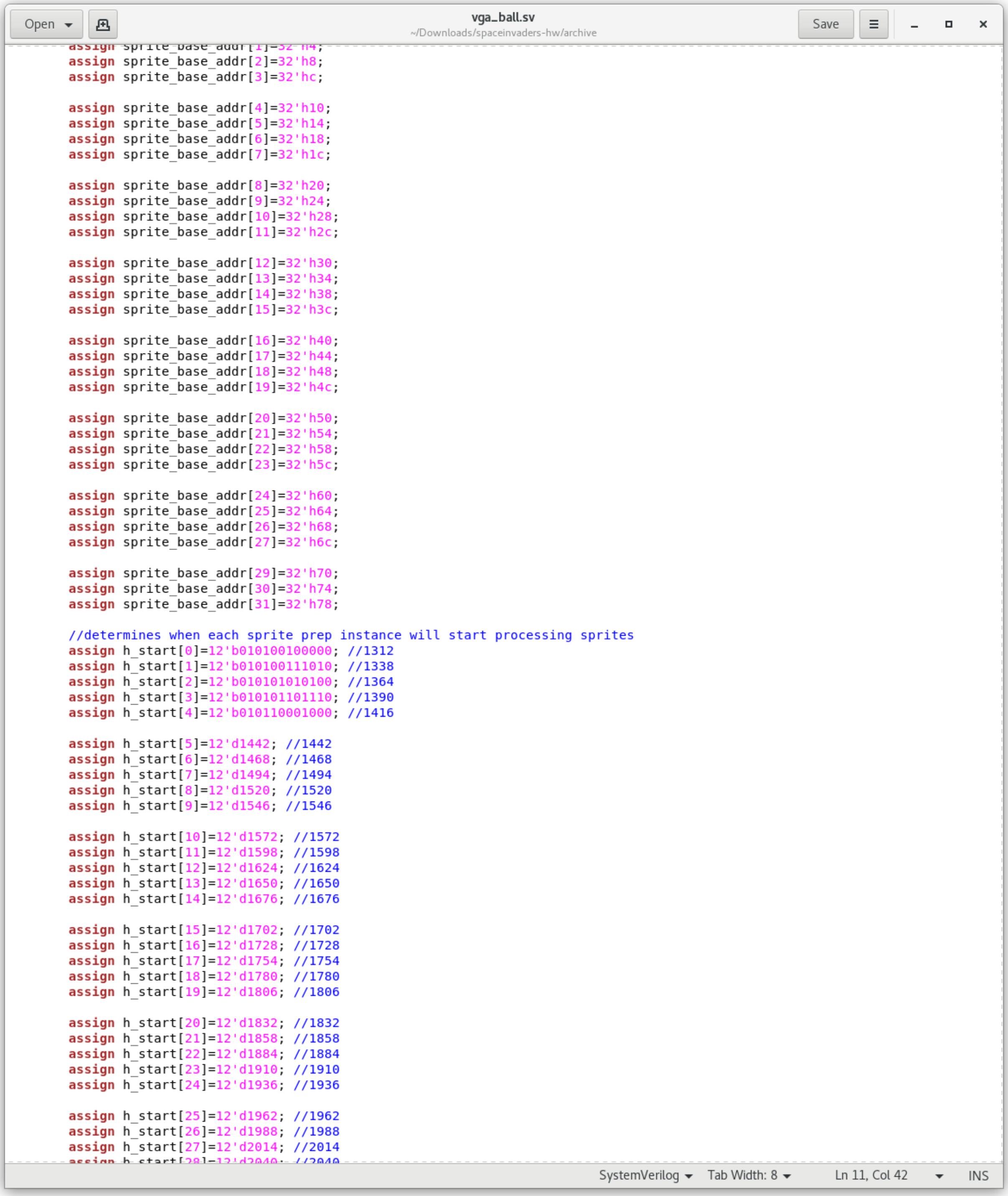
Alan Hwang	Game Hardware & Basic Software
Zach Burpee	Game Software & Basic Hardware
Mili Sehgal	Audio Hardware & Basic Software

Code Screen Shots - Hardware

vga_ball.sv



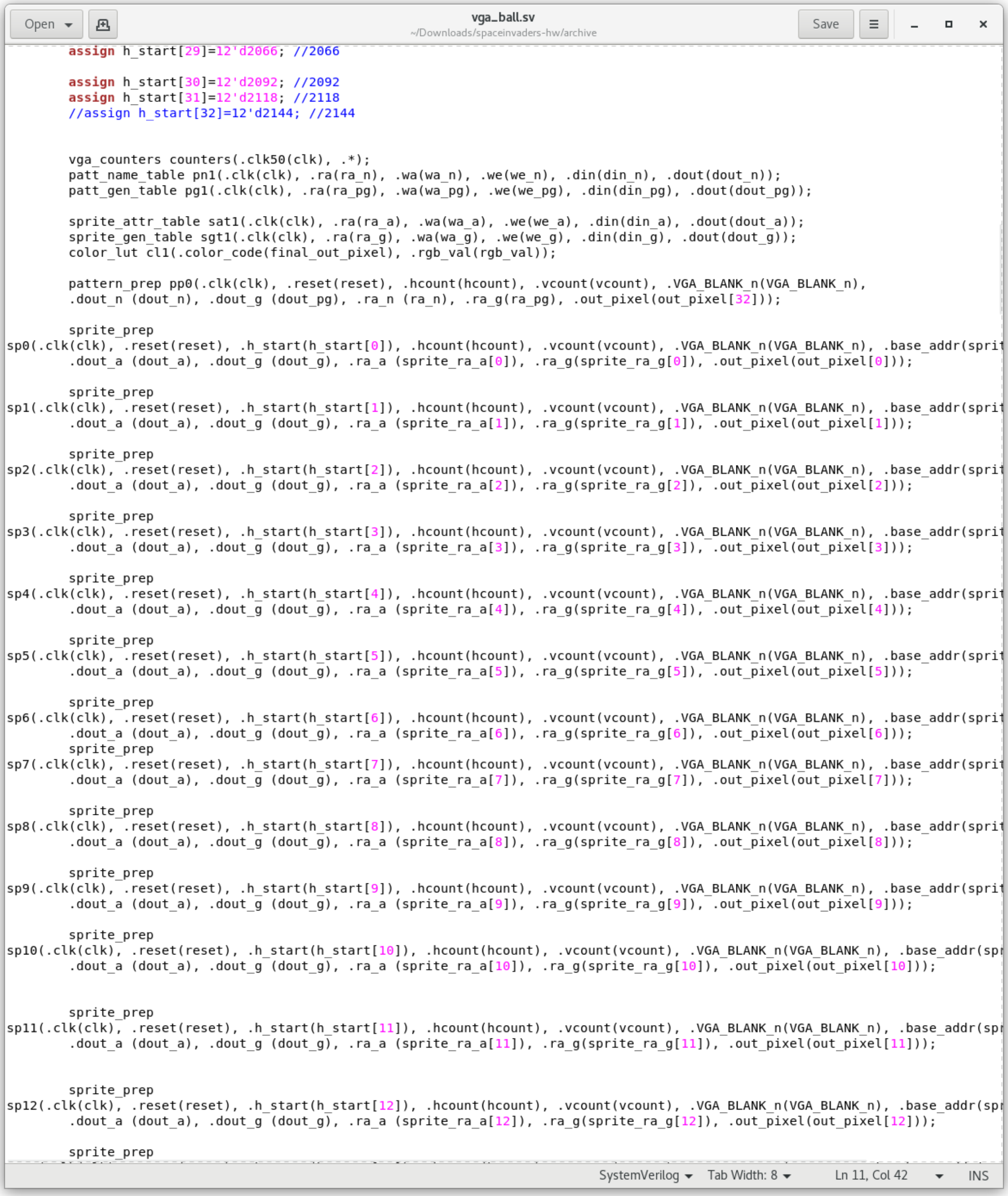
```
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module vga_ball(input logic        clk,
                input logic        reset,
                input logic [31:0]  writedata,
                input logic        write,
                input              chipselect,
                input logic [3:0]  address,

                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 [3:0] out_pixel[32:0]; //output pixels values from each of 32 sprites + 1 pattern
        logic [3:0] final_out_pixel; //actual output pixel to display
        logic [7:0]        background_r, background_g, background_b;
        logic [23:0] rgb_val; //final RGB value to display


        //for pattern name table
        logic [11:0] ra_n, wa_n; //12 bits
        logic we_n;
        logic [7:0] din_n;
        logic [7:0] dout_n;

        //for pattern generator table
        logic [10:0] ra_pg, wa_pg; //change later
        logic we_pg;
        logic [7:0] din_pg;
        logic [7:0] dout_pg;

        //for sprite attribute table
        logic [31:0] ra_a, wa_a;  //32  simultaneous sprites
        logic we_a;
        logic [7:0] din_a;
        logic [7:0] dout_a;

        //for sprite generator table
        logic [11:0] ra_g, wa_g; //32*128 sprite -> 12 bit addr
        logic we_g;
        logic [7:0] din_g;
        logic [7:0] dout_g;

        logic [31:0] sprite_base_addr[31:0]; //sprite attr table base address
        logic [11:0] h_start[31:0]; //hcount at which sprite_prep n starts
        logic [31:0] sprite_ra_a[31:0]; //requested read address for sprite attr table from sprite prep modules
        logic [11:0] sprite_ra_g[31:0]; //requested read address for sprite gen table from sprite prep modules


        //determines where each sprite prep instance will start reading the attr table from
        assign sprite_base_addr[0]=32'h0;
        assign sprite_base_addr[1]=32'h4;
        assign sprite_base_addr[2]=32'h8;
        assign sprite_base_addr[3]=32'hc;

        assign sprite_base_addr[4]=32'h10;
        assign sprite_base_addr[5]=32'h14;
        assign sprite_base_addr[6]=32'h18;
        assign sprite_base_addr[7]=32'h1c;

        assign sprite_base_addr[8]=32'h20;
        assign sprite_base_addr[9]=32'h24;
        assign sprite_base_addr[10]=32'h28;
        assign sprite_base_addr[11]=32'h2c;
```



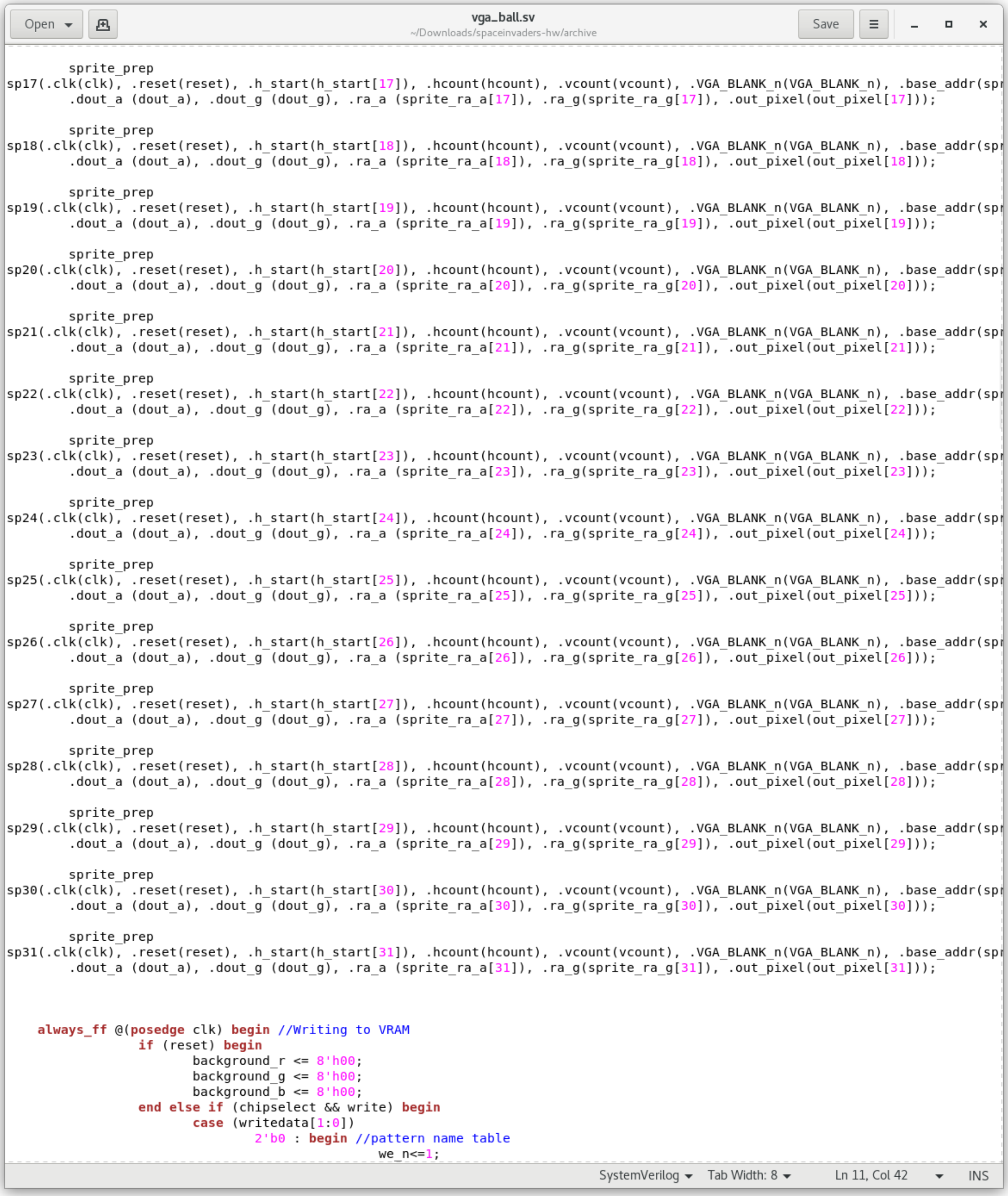
vga_ball.sv
~/Downloads/spaceinvaders-hw/archive

```
        assign sprite_base_addr[1]=32'h4;
        assign sprite_base_addr[2]=32'h8;
        assign sprite_base_addr[3]=32'hc;

        assign sprite_base_addr[4]=32'h10;
        assign sprite_base_addr[5]=32'h14;
        assign sprite_base_addr[6]=32'h18;
        assign sprite_base_addr[7]=32'h1c;

        assign sprite_base_addr[8]=32'h20;
        assign sprite_base_addr[9]=32'h24;
        assign sprite_base_addr[10]=32'h28;
        assign sprite_base_addr[11]=32'h2c;

        assign sprite_base_addr[12]=32'h30;
        assign sprite_base_addr[13]=32'h34;
        assign sprite_base_addr[14]=32'h38;
        assign sprite_base_addr[15]=32'h3c;

        assign sprite_base_addr[16]=32'h40;
        assign sprite_base_addr[17]=32'h44;
        assign sprite_base_addr[18]=32'h48;
        assign sprite_base_addr[19]=32'h4c;

        assign sprite_base_addr[20]=32'h50;
        assign sprite_base_addr[21]=32'h54;
        assign sprite_base_addr[22]=32'h58;
        assign sprite_base_addr[23]=32'h5c;

        assign sprite_base_addr[24]=32'h60;
        assign sprite_base_addr[25]=32'h64;
        assign sprite_base_addr[26]=32'h68;
        assign sprite_base_addr[27]=32'h6c;

        assign sprite_base_addr[29]=32'h70;
        assign sprite_base_addr[30]=32'h74;
        assign sprite_base_addr[31]=32'h78;

        //determines when each sprite prep instance will start processing sprites
        assign h_start[0]=12'b010100100000; //1312
        assign h_start[1]=12'b010100111010; //1338
        assign h_start[2]=12'b010101010100; //1364
        assign h_start[3]=12'b010101101110; //1390
        assign h_start[4]=12'b010110001000; //1416

        assign h_start[5]=12'd1442; //1442
        assign h_start[6]=12'd1468; //1468
        assign h_start[7]=12'd1494; //1494
        assign h_start[8]=12'd1520; //1520
        assign h_start[9]=12'd1546; //1546

        assign h_start[10]=12'd1572; //1572
        assign h_start[11]=12'd1598; //1598
        assign h_start[12]=12'd1624; //1624
        assign h_start[13]=12'd1650; //1650
        assign h_start[14]=12'd1676; //1676

        assign h_start[15]=12'd1702; //1702
        assign h_start[16]=12'd1728; //1728
        assign h_start[17]=12'd1754; //1754
        assign h_start[18]=12'd1780; //1780
        assign h_start[19]=12'd1806; //1806

        assign h_start[20]=12'd1832; //1832
        assign h_start[21]=12'd1858; //1858
        assign h_start[22]=12'd1884; //1884
        assign h_start[23]=12'd1910; //1910
        assign h_start[24]=12'd1936; //1936

        assign h_start[25]=12'd1962; //1962
        assign h_start[26]=12'd1988; //1988
        assign h_start[27]=12'd2014; //2014
        assign h_start[28]=12'd2040; //2040
```

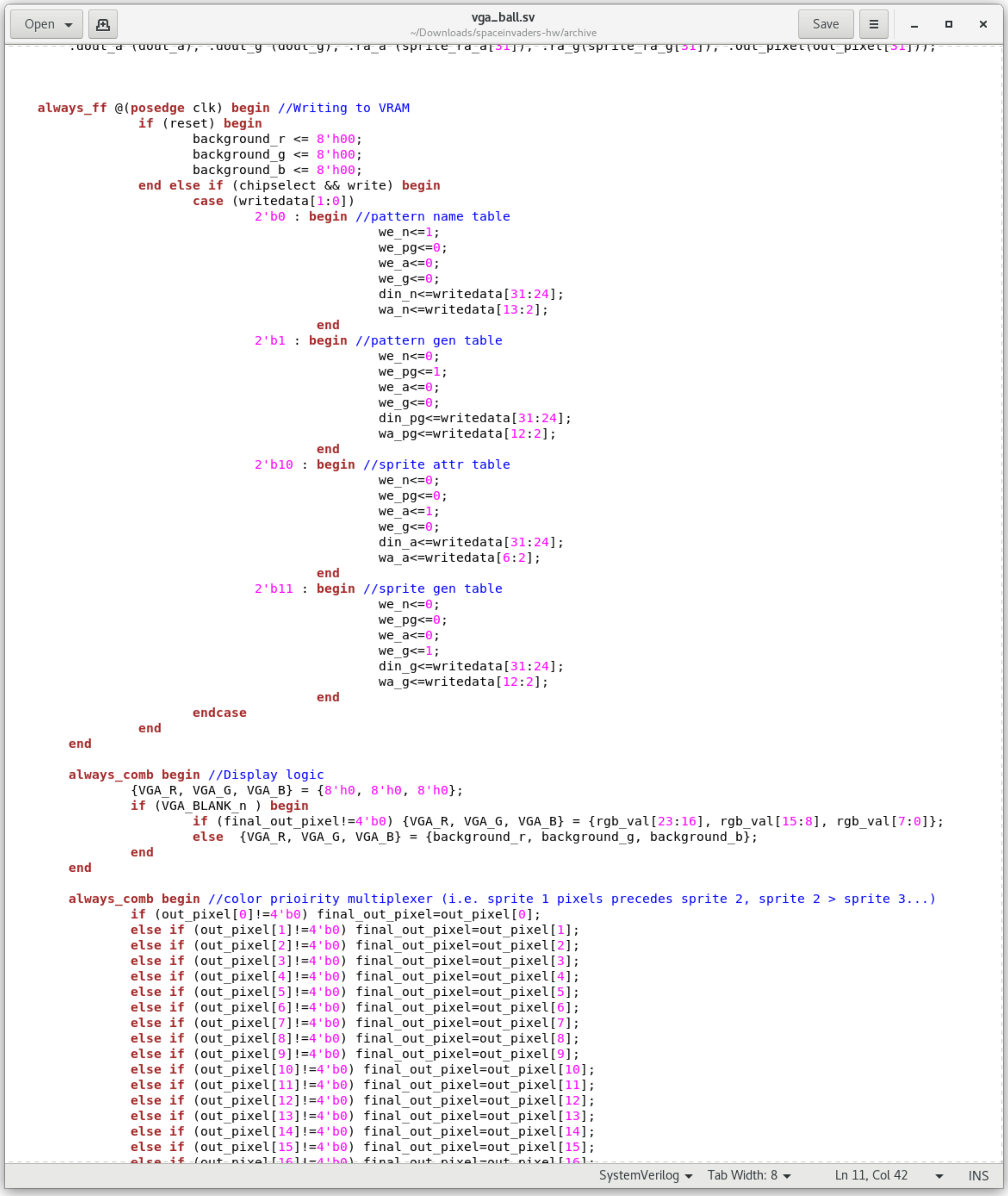
SystemVerilog   Tab Width: 8   Ln 11, Col 42   INS



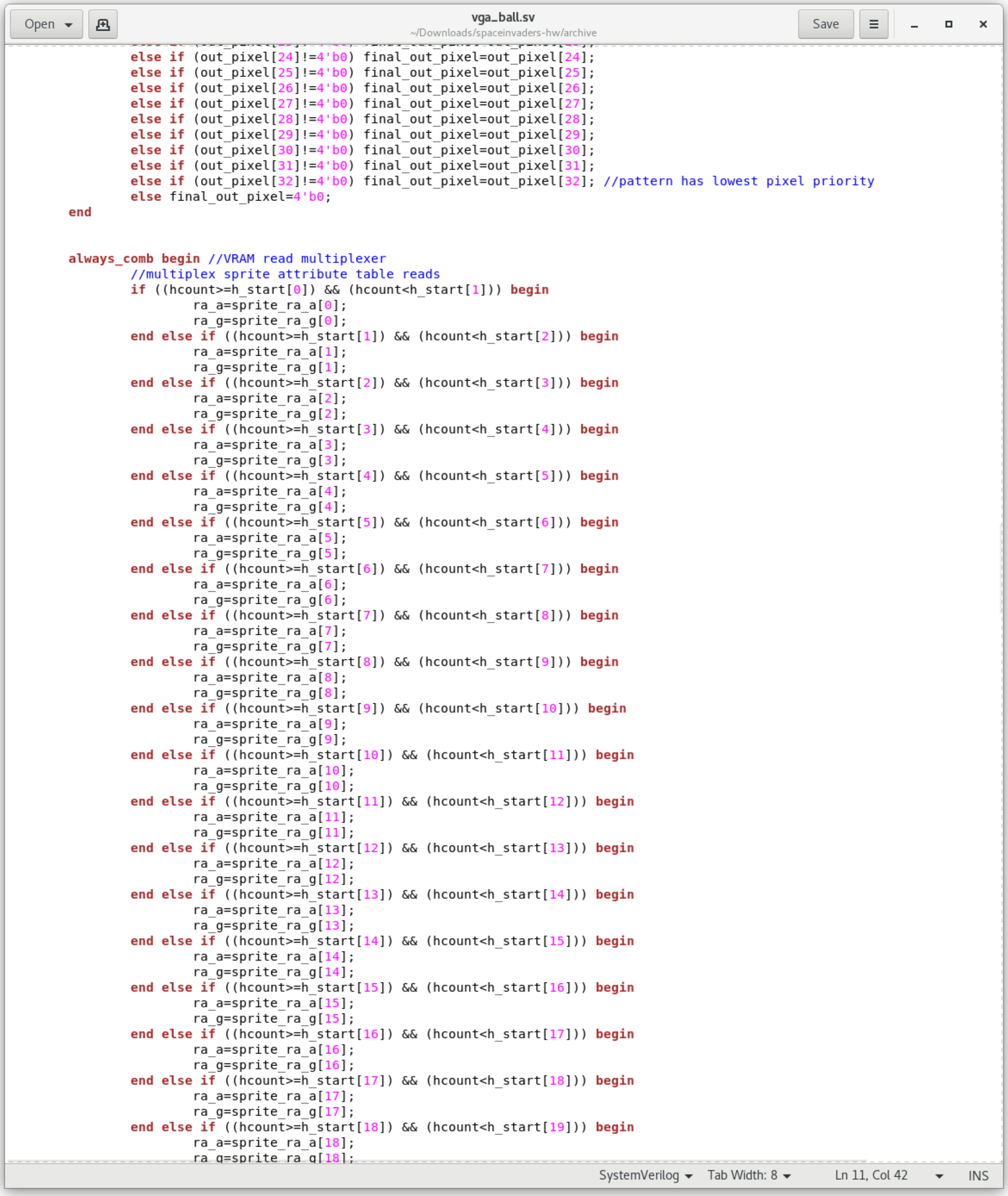
```
        assign h_start[29]=12'd2066; //2066

        assign h_start[30]=12'd2092; //2092
        assign h_start[31]=12'd2118; //2118
        //assign h_start[32]=12'd2144; //2144


        vga_counters counters(.clk50(clk), .*);
        patt_name_table pn1(.clk(clk), .ra(ra_n), .wa(wa_n), .we(we_n), .din(din_n), .dout(dout_n));
        patt_gen_table pg1(.clk(clk), .ra(ra_pg), .wa(wa_pg), .we(we_pg), .din(din_pg), .dout(dout_pg));

        sprite_attr_table sat1(.clk(clk), .ra(ra_a), .wa(wa_a), .we(we_a), .din(din_a), .dout(dout_a));
        sprite_gen_table sgt1(.clk(clk), .ra(ra_g), .wa(wa_g), .we(we_g), .din(din_g), .dout(dout_g));
        color_lut cl1(.color_code(final_out_pixel), .rgb_val(rgb_val));

        pattern_prep pp0(.clk(clk), .reset(reset), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n),
        .dout_n (dout_n), .dout_g (dout_pg), .ra_n (ra_n), .ra_g(ra_pg), .out_pixel(out_pixel[32]));

        sprite_prep
sp0(.clk(clk), .reset(reset), .h_start(h_start[0]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[0]), .ra_g(sprite_ra_g[0]), .out_pixel(out_pixel[0]));

        sprite_prep
sp1(.clk(clk), .reset(reset), .h_start(h_start[1]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[1]), .ra_g(sprite_ra_g[1]), .out_pixel(out_pixel[1]));

        sprite_prep
sp2(.clk(clk), .reset(reset), .h_start(h_start[2]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[2]), .ra_g(sprite_ra_g[2]), .out_pixel(out_pixel[2]));

        sprite_prep
sp3(.clk(clk), .reset(reset), .h_start(h_start[3]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[3]), .ra_g(sprite_ra_g[3]), .out_pixel(out_pixel[3]));

        sprite_prep
sp4(.clk(clk), .reset(reset), .h_start(h_start[4]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[4]), .ra_g(sprite_ra_g[4]), .out_pixel(out_pixel[4]));

        sprite_prep
sp5(.clk(clk), .reset(reset), .h_start(h_start[5]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[5]), .ra_g(sprite_ra_g[5]), .out_pixel(out_pixel[5]));

        sprite_prep
sp6(.clk(clk), .reset(reset), .h_start(h_start[6]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[6]), .ra_g(sprite_ra_g[6]), .out_pixel(out_pixel[6]));
        sprite_prep
sp7(.clk(clk), .reset(reset), .h_start(h_start[7]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[7]), .ra_g(sprite_ra_g[7]), .out_pixel(out_pixel[7]));

        sprite_prep
sp8(.clk(clk), .reset(reset), .h_start(h_start[8]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[8]), .ra_g(sprite_ra_g[8]), .out_pixel(out_pixel[8]));

        sprite_prep
sp9(.clk(clk), .reset(reset), .h_start(h_start[9]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(sprit
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[9]), .ra_g(sprite_ra_g[9]), .out_pixel(out_pixel[9]));

        sprite_prep
sp10(.clk(clk), .reset(reset), .h_start(h_start[10]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[10]), .ra_g(sprite_ra_g[10]), .out_pixel(out_pixel[10]));


        sprite_prep
sp11(.clk(clk), .reset(reset), .h_start(h_start[11]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[11]), .ra_g(sprite_ra_g[11]), .out_pixel(out_pixel[11]));


        sprite_prep
sp12(.clk(clk), .reset(reset), .h_start(h_start[12]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[12]), .ra_g(sprite_ra_g[12]), .out_pixel(out_pixel[12]));

        sprite_prep
```



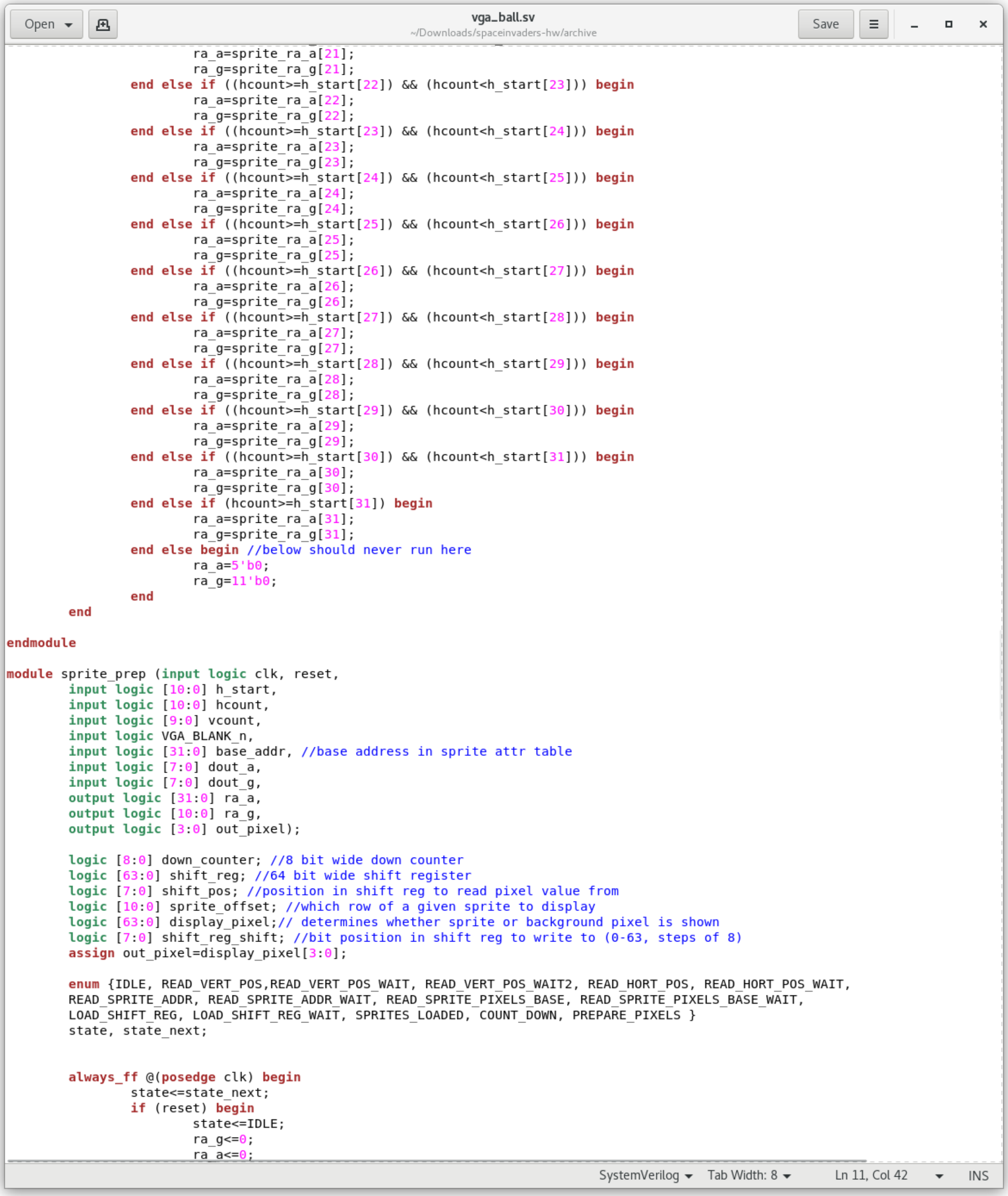
```
        sprite_prep
sp17(.clk(clk), .reset(reset), .h_start(h_start[17]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[17]), .ra_g(sprite_ra_g[17]), .out_pixel(out_pixel[17]));

        sprite_prep
sp18(.clk(clk), .reset(reset), .h_start(h_start[18]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[18]), .ra_g(sprite_ra_g[18]), .out_pixel(out_pixel[18]));

        sprite_prep
sp19(.clk(clk), .reset(reset), .h_start(h_start[19]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[19]), .ra_g(sprite_ra_g[19]), .out_pixel(out_pixel[19]));

        sprite_prep
sp20(.clk(clk), .reset(reset), .h_start(h_start[20]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[20]), .ra_g(sprite_ra_g[20]), .out_pixel(out_pixel[20]));

        sprite_prep
sp21(.clk(clk), .reset(reset), .h_start(h_start[21]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[21]), .ra_g(sprite_ra_g[21]), .out_pixel(out_pixel[21]));

        sprite_prep
sp22(.clk(clk), .reset(reset), .h_start(h_start[22]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[22]), .ra_g(sprite_ra_g[22]), .out_pixel(out_pixel[22]));

        sprite_prep
sp23(.clk(clk), .reset(reset), .h_start(h_start[23]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[23]), .ra_g(sprite_ra_g[23]), .out_pixel(out_pixel[23]));

        sprite_prep
sp24(.clk(clk), .reset(reset), .h_start(h_start[24]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[24]), .ra_g(sprite_ra_g[24]), .out_pixel(out_pixel[24]));

        sprite_prep
sp25(.clk(clk), .reset(reset), .h_start(h_start[25]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[25]), .ra_g(sprite_ra_g[25]), .out_pixel(out_pixel[25]));

        sprite_prep
sp26(.clk(clk), .reset(reset), .h_start(h_start[26]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[26]), .ra_g(sprite_ra_g[26]), .out_pixel(out_pixel[26]));

        sprite_prep
sp27(.clk(clk), .reset(reset), .h_start(h_start[27]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[27]), .ra_g(sprite_ra_g[27]), .out_pixel(out_pixel[27]));

        sprite_prep
sp28(.clk(clk), .reset(reset), .h_start(h_start[28]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[28]), .ra_g(sprite_ra_g[28]), .out_pixel(out_pixel[28]));

        sprite_prep
sp29(.clk(clk), .reset(reset), .h_start(h_start[29]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[29]), .ra_g(sprite_ra_g[29]), .out_pixel(out_pixel[29]));

        sprite_prep
sp30(.clk(clk), .reset(reset), .h_start(h_start[30]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[30]), .ra_g(sprite_ra_g[30]), .out_pixel(out_pixel[30]));

        sprite_prep
sp31(.clk(clk), .reset(reset), .h_start(h_start[31]), .hcount(hcount), .vcount(vcount), .VGA_BLANK_n(VGA_BLANK_n), .base_addr(spr
        .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[31]), .ra_g(sprite_ra_g[31]), .out_pixel(out_pixel[31]));



    always_ff @(posedge clk) begin //Writing to VRAM
                 if (reset) begin
                        background_r <= 8'h00;
                        background_g <= 8'h00;
                        background_b <= 8'h00;
                 end else if (chipselect && write) begin
                        case (writedata[1:0])
                                2'b0 : begin //pattern name table
                                                we_n<=1;
```



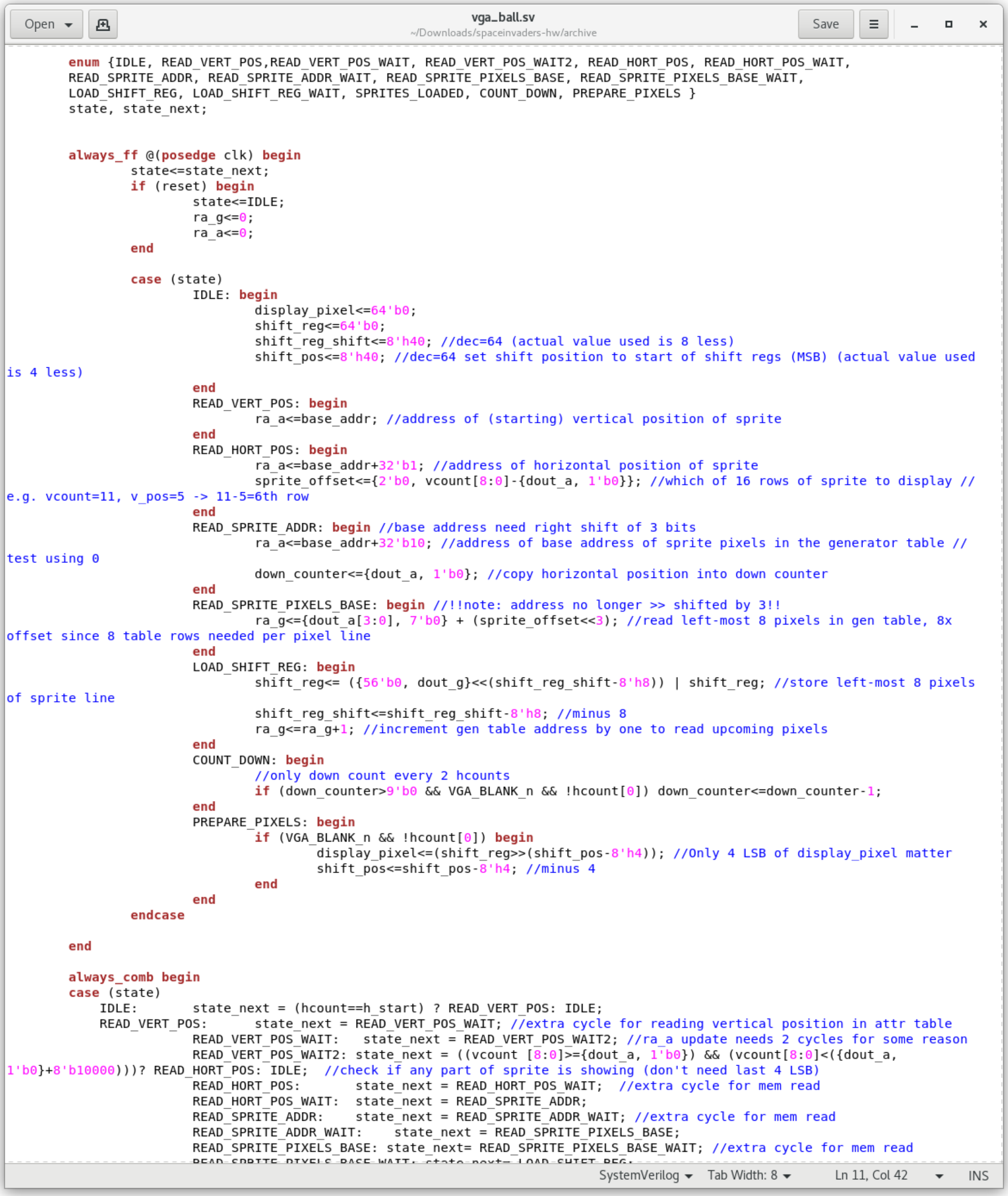
```
    .dout_a (dout_a), .dout_g (dout_g), .ra_a (sprite_ra_a[31]), .ra_g(sprite_ra_g[31]), .out_pixel(out_pixel[31]));



always_ff @(posedge clk) begin //Writing to VRAM
             if (reset) begin
                    background_r <= 8'h00;
                    background_g <= 8'h00;
                    background_b <= 8'h00;
             end else if (chipselect && write) begin
                    case (writedata[1:0])
                            2'b0 : begin //pattern name table
                                            we_n<=1;
                                            we_pg<=0;
                                            we_a<=0;
                                            we_g<=0;
                                            din_n<=writedata[31:24];
                                            wa_n<=writedata[13:2];
                                    end
                            2'b1 : begin //pattern gen table
                                            we_n<=0;
                                            we_pg<=1;
                                            we_a<=0;
                                            we_g<=0;
                                            din_pg<=writedata[31:24];
                                            wa_pg<=writedata[12:2];
                                    end
                            2'b10 : begin //sprite attr table
                                            we_n<=0;
                                            we_pg<=0;
                                            we_a<=1;
                                            we_g<=0;
                                            din_a<=writedata[31:24];
                                            wa_a<=writedata[6:2];
                                    end
                            2'b11 : begin //sprite gen table
                                            we_n<=0;
                                            we_pg<=0;
                                            we_a<=0;
                                            we_g<=1;
                                            din_g<=writedata[31:24];
                                            wa_g<=writedata[12:2];
                                    end
                    endcase
             end
    end

    always_comb begin //Display logic
            {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
            if (VGA_BLANK_n ) begin
                    if (final_out_pixel!=4'b0) {VGA_R, VGA_G, VGA_B} = {rgb_val[23:16], rgb_val[15:8], rgb_val[7:0]};
                    else  {VGA_R, VGA_G, VGA_B} = {background_r, background_g, background_b};
            end
    end

    always_comb begin //color prioirity multiplexer (i.e. sprite 1 pixels precedes sprite 2, sprite 2 > sprite 3...)
            if (out_pixel[0]!=4'b0) final_out_pixel=out_pixel[0];
            else if (out_pixel[1]!=4'b0) final_out_pixel=out_pixel[1];
            else if (out_pixel[2]!=4'b0) final_out_pixel=out_pixel[2];
            else if (out_pixel[3]!=4'b0) final_out_pixel=out_pixel[3];
            else if (out_pixel[4]!=4'b0) final_out_pixel=out_pixel[4];
            else if (out_pixel[5]!=4'b0) final_out_pixel=out_pixel[5];
            else if (out_pixel[6]!=4'b0) final_out_pixel=out_pixel[6];
            else if (out_pixel[7]!=4'b0) final_out_pixel=out_pixel[7];
            else if (out_pixel[8]!=4'b0) final_out_pixel=out_pixel[8];
            else if (out_pixel[9]!=4'b0) final_out_pixel=out_pixel[9];
            else if (out_pixel[10]!=4'b0) final_out_pixel=out_pixel[10];
            else if (out_pixel[11]!=4'b0) final_out_pixel=out_pixel[11];
            else if (out_pixel[12]!=4'b0) final_out_pixel=out_pixel[12];
            else if (out_pixel[13]!=4'b0) final_out_pixel=out_pixel[13];
            else if (out_pixel[14]!=4'b0) final_out_pixel=out_pixel[14];
            else if (out_pixel[15]!=4'b0) final_out_pixel=out_pixel[15];
            else if (out_pixel[16]!=4'b0) final_out_pixel=out_pixel[16];
```



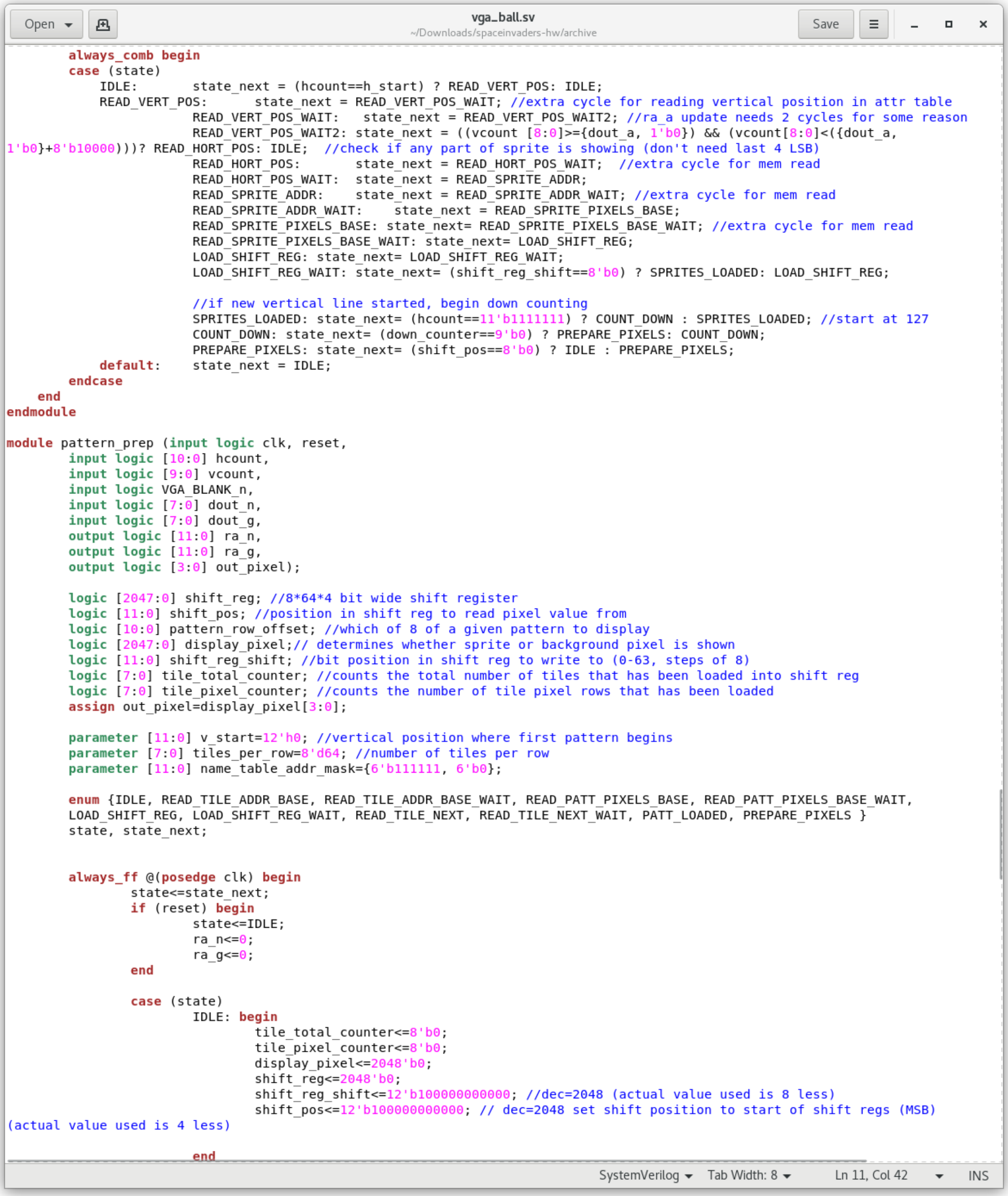
```
                else if (out_pixel[24]!=4'b0) final_out_pixel=out_pixel[24];
                else if (out_pixel[25]!=4'b0) final_out_pixel=out_pixel[25];
                else if (out_pixel[26]!=4'b0) final_out_pixel=out_pixel[26];
                else if (out_pixel[27]!=4'b0) final_out_pixel=out_pixel[27];
                else if (out_pixel[28]!=4'b0) final_out_pixel=out_pixel[28];
                else if (out_pixel[29]!=4'b0) final_out_pixel=out_pixel[29];
                else if (out_pixel[30]!=4'b0) final_out_pixel=out_pixel[30];
                else if (out_pixel[31]!=4'b0) final_out_pixel=out_pixel[31];
                else if (out_pixel[32]!=4'b0) final_out_pixel=out_pixel[32]; //pattern has lowest pixel priority
                else final_out_pixel=4'b0;
        end


        always_comb begin //VRAM read multiplexer
                //multiplex sprite attribute table reads
                if ((hcount>=h_start[0]) && (hcount<h_start[1])) begin
                        ra_a=sprite_ra_a[0];
                        ra_g=sprite_ra_g[0];
                end else if ((hcount>=h_start[1]) && (hcount<h_start[2])) begin
                        ra_a=sprite_ra_a[1];
                        ra_g=sprite_ra_g[1];
                end else if ((hcount>=h_start[2]) && (hcount<h_start[3])) begin
                        ra_a=sprite_ra_a[2];
                        ra_g=sprite_ra_g[2];
                end else if ((hcount>=h_start[3]) && (hcount<h_start[4])) begin
                        ra_a=sprite_ra_a[3];
                        ra_g=sprite_ra_g[3];
                end else if ((hcount>=h_start[4]) && (hcount<h_start[5])) begin
                        ra_a=sprite_ra_a[4];
                        ra_g=sprite_ra_g[4];
                end else if ((hcount>=h_start[5]) && (hcount<h_start[6])) begin
                        ra_a=sprite_ra_a[5];
                        ra_g=sprite_ra_g[5];
                end else if ((hcount>=h_start[6]) && (hcount<h_start[7])) begin
                        ra_a=sprite_ra_a[6];
                        ra_g=sprite_ra_g[6];
                end else if ((hcount>=h_start[7]) && (hcount<h_start[8])) begin
                        ra_a=sprite_ra_a[7];
                        ra_g=sprite_ra_g[7];
                end else if ((hcount>=h_start[8]) && (hcount<h_start[9])) begin
                        ra_a=sprite_ra_a[8];
                        ra_g=sprite_ra_g[8];
                end else if ((hcount>=h_start[9]) && (hcount<h_start[10])) begin
                        ra_a=sprite_ra_a[9];
                        ra_g=sprite_ra_g[9];
                end else if ((hcount>=h_start[10]) && (hcount<h_start[11])) begin
                        ra_a=sprite_ra_a[10];
                        ra_g=sprite_ra_g[10];
                end else if ((hcount>=h_start[11]) && (hcount<h_start[12])) begin
                        ra_a=sprite_ra_a[11];
                        ra_g=sprite_ra_g[11];
                end else if ((hcount>=h_start[12]) && (hcount<h_start[13])) begin
                        ra_a=sprite_ra_a[12];
                        ra_g=sprite_ra_g[12];
                end else if ((hcount>=h_start[13]) && (hcount<h_start[14])) begin
                        ra_a=sprite_ra_a[13];
                        ra_g=sprite_ra_g[13];
                end else if ((hcount>=h_start[14]) && (hcount<h_start[15])) begin
                        ra_a=sprite_ra_a[14];
                        ra_g=sprite_ra_g[14];
                end else if ((hcount>=h_start[15]) && (hcount<h_start[16])) begin
                        ra_a=sprite_ra_a[15];
                        ra_g=sprite_ra_g[15];
                end else if ((hcount>=h_start[16]) && (hcount<h_start[17])) begin
                        ra_a=sprite_ra_a[16];
                        ra_g=sprite_ra_g[16];
                end else if ((hcount>=h_start[17]) && (hcount<h_start[18])) begin
                        ra_a=sprite_ra_a[17];
                        ra_g=sprite_ra_g[17];
                end else if ((hcount>=h_start[18]) && (hcount<h_start[19])) begin
                        ra_a=sprite_ra_a[18];
                        ra_g=sprite_ra_g[18];
```



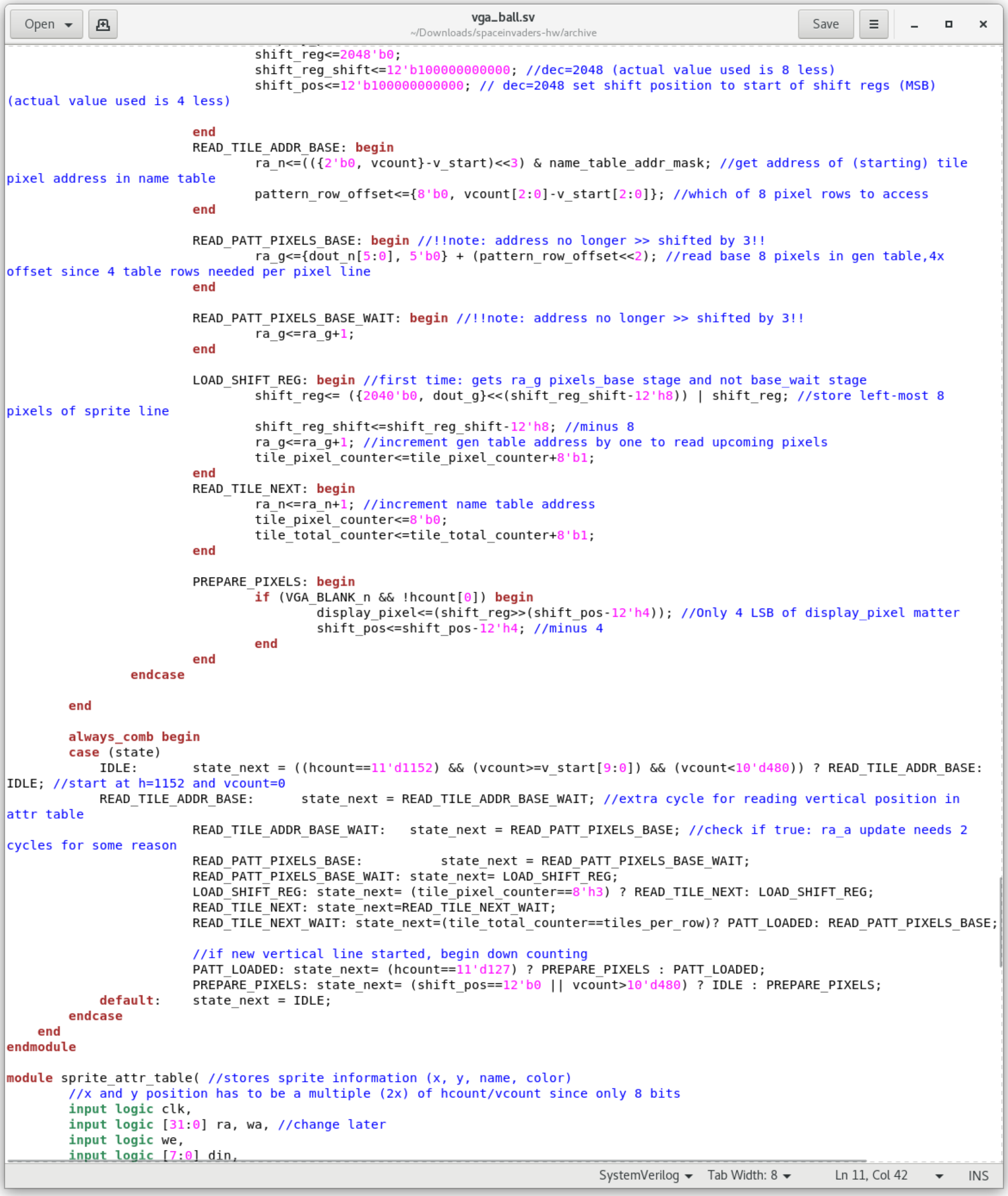
```
                        ra_a=sprite_ra_a[21];
                        ra_g=sprite_ra_g[21];
                end else if ((hcount>=h_start[22]) && (hcount<h_start[23])) begin
                        ra_a=sprite_ra_a[22];
                        ra_g=sprite_ra_g[22];
                end else if ((hcount>=h_start[23]) && (hcount<h_start[24])) begin
                        ra_a=sprite_ra_a[23];
                        ra_g=sprite_ra_g[23];
                end else if ((hcount>=h_start[24]) && (hcount<h_start[25])) begin
                        ra_a=sprite_ra_a[24];
                        ra_g=sprite_ra_g[24];
                end else if ((hcount>=h_start[25]) && (hcount<h_start[26])) begin
                        ra_a=sprite_ra_a[25];
                        ra_g=sprite_ra_g[25];
                end else if ((hcount>=h_start[26]) && (hcount<h_start[27])) begin
                        ra_a=sprite_ra_a[26];
                        ra_g=sprite_ra_g[26];
                end else if ((hcount>=h_start[27]) && (hcount<h_start[28])) begin
                        ra_a=sprite_ra_a[27];
                        ra_g=sprite_ra_g[27];
                end else if ((hcount>=h_start[28]) && (hcount<h_start[29])) begin
                        ra_a=sprite_ra_a[28];
                        ra_g=sprite_ra_g[28];
                end else if ((hcount>=h_start[29]) && (hcount<h_start[30])) begin
                        ra_a=sprite_ra_a[29];
                        ra_g=sprite_ra_g[29];
                end else if ((hcount>=h_start[30]) && (hcount<h_start[31])) begin
                        ra_a=sprite_ra_a[30];
                        ra_g=sprite_ra_g[30];
                end else if (hcount>=h_start[31]) begin
                        ra_a=sprite_ra_a[31];
                        ra_g=sprite_ra_g[31];
                end else begin //below should never run here
                        ra_a=5'b0;
                        ra_g=11'b0;
                end
        end

endmodule

module sprite_prep (input logic clk, reset,
        input logic [10:0] h_start,
        input logic [10:0] hcount,
        input logic [9:0] vcount,
        input logic VGA_BLANK_n,
        input logic [31:0] base_addr, //base address in sprite attr table
        input logic [7:0] dout_a,
        input logic [7:0] dout_g,
        output logic [31:0] ra_a,
        output logic [10:0] ra_g,
        output logic [3:0] out_pixel);

        logic [8:0] down_counter; //8 bit wide down counter
        logic [63:0] shift_reg; //64 bit wide shift register
        logic [7:0] shift_pos; //position in shift reg to read pixel value from
        logic [10:0] sprite_offset; //which row of a given sprite to display
        logic [63:0] display_pixel;// determines whether sprite or background pixel is shown
        logic [7:0] shift_reg_shift; //bit position in shift reg to write to (0-63, steps of 8)
        assign out_pixel=display_pixel[3:0];

        enum {IDLE, READ_VERT_POS,READ_VERT_POS_WAIT, READ_VERT_POS_WAIT2, READ_HORT_POS, READ_HORT_POS_WAIT,
        READ_SPRITE_ADDR, READ_SPRITE_ADDR_WAIT, READ_SPRITE_PIXELS_BASE, READ_SPRITE_PIXELS_BASE_WAIT,
        LOAD_SHIFT_REG, LOAD_SHIFT_REG_WAIT, SPRITES_LOADED, COUNT_DOWN, PREPARE_PIXELS }
        state, state_next;


        always_ff @(posedge clk) begin
                state<=state_next;
                if (reset) begin
                        state<=IDLE;
                        ra_g<=0;
                        ra_a<=0;
```



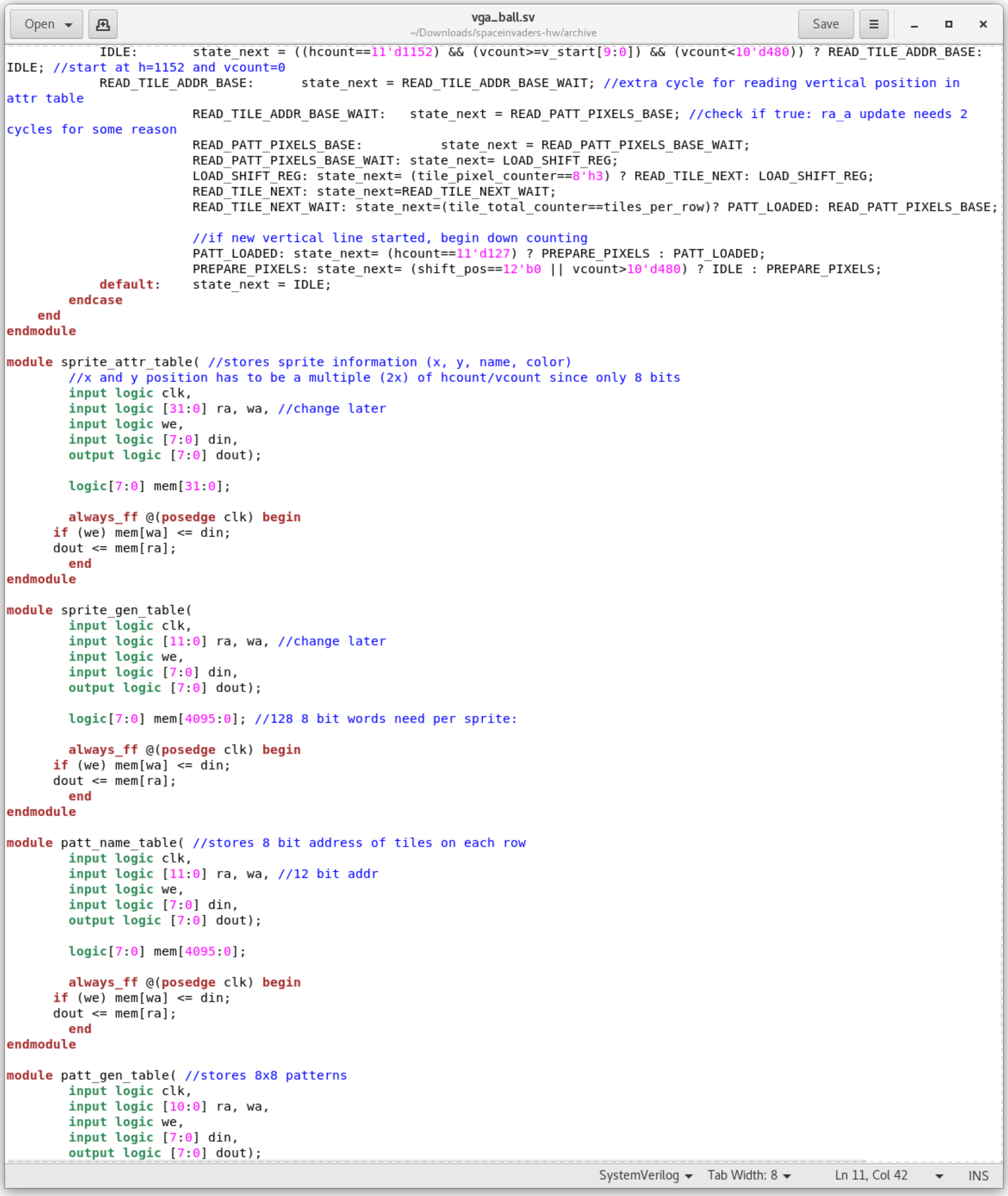
```
        enum {IDLE, READ_VERT_POS,READ_VERT_POS_WAIT, READ_VERT_POS_WAIT2, READ_HORT_POS, READ_HORT_POS_WAIT,
        READ_SPRITE_ADDR, READ_SPRITE_ADDR_WAIT, READ_SPRITE_PIXELS_BASE, READ_SPRITE_PIXELS_BASE_WAIT,
        LOAD_SHIFT_REG, LOAD_SHIFT_REG_WAIT, SPRITES_LOADED, COUNT_DOWN, PREPARE_PIXELS }
        state, state_next;


        always_ff @(posedge clk) begin
                state<=state_next;
                if (reset) begin
                        state<=IDLE;
                        ra_g<=0;
                        ra_a<=0;
                end

                case (state)
                        IDLE: begin
                                display_pixel<=64'b0;
                                shift_reg<=64'b0;
                                shift_reg_shift<=8'h40; //dec=64 (actual value used is 8 less)
                                shift_pos<=8'h40; //dec=64 set shift position to start of shift regs (MSB) (actual value used
is 4 less)
                        end
                        READ_VERT_POS: begin
                                ra_a<=base_addr; //address of (starting) vertical position of sprite
                        end
                        READ_HORT_POS: begin
                                ra_a<=base_addr+32'b1; //address of horizontal position of sprite
                                sprite_offset<={2'b0, vcount[8:0]-{dout_a, 1'b0}}; //which of 16 rows of sprite to display //
e.g. vcount=11, v_pos=5 -> 11-5=6th row
                        end
                        READ_SPRITE_ADDR: begin //base address need right shift of 3 bits
                                ra_a<=base_addr+32'b10; //address of base address of sprite pixels in the generator table //
test using 0

                                down_counter<={dout_a, 1'b0}; //copy horizontal position into down counter
                        end
                        READ_SPRITE_PIXELS_BASE: begin //!!note: address no longer >> shifted by 3!!
                                ra_g<={dout_a[3:0], 7'b0} + (sprite_offset<<3); //read left-most 8 pixels in gen table, 8x
offset since 8 table rows needed per pixel line
                        end
                        LOAD_SHIFT_REG: begin
                                shift_reg<= ({56'b0, dout_g}<<(shift_reg_shift-8'h8)) | shift_reg; //store left-most 8 pixels
of sprite line

                                shift_reg_shift<=shift_reg_shift-8'h8; //minus 8
                                ra_g<=ra_g+1; //increment gen table address by one to read upcoming pixels
                        end
                        COUNT_DOWN: begin
                                //only down count every 2 hcounts
                                if (down_counter>9'b0 && VGA_BLANK_n && !hcount[0]) down_counter<=down_counter-1;
                        end
                        PREPARE_PIXELS: begin
                                if (VGA_BLANK_n && !hcount[0]) begin
                                        display_pixel<=(shift_reg>>(shift_pos-8'h4)); //Only 4 LSB of display_pixel matter
                                        shift_pos<=shift_pos-8'h4; //minus 4
                                end
                        end
                endcase

        end

        always_comb begin
        case (state)
            IDLE:       state_next = (hcount==h_start) ? READ_VERT_POS: IDLE;
            READ_VERT_POS:      state_next = READ_VERT_POS_WAIT; //extra cycle for reading vertical position in attr table
                        READ_VERT_POS_WAIT:   state_next = READ_VERT_POS_WAIT2; //ra_a update needs 2 cycles for some reason
                        READ_VERT_POS_WAIT2: state_next = ((vcount [8:0]>={dout_a, 1'b0}) && (vcount[8:0]<({dout_a,
1'b0}+8'b10000)))? READ_HORT_POS: IDLE;  //check if any part of sprite is showing (don't need last 4 LSB)
                        READ_HORT_POS:       state_next = READ_HORT_POS_WAIT;  //extra cycle for mem read
                        READ_HORT_POS_WAIT:  state_next = READ_SPRITE_ADDR;
                        READ_SPRITE_ADDR:    state_next = READ_SPRITE_ADDR_WAIT; //extra cycle for mem read
                        READ_SPRITE_ADDR_WAIT:    state_next = READ_SPRITE_PIXELS_BASE;
                        READ_SPRITE_PIXELS_BASE: state_next= READ_SPRITE_PIXELS_BASE_WAIT; //extra cycle for mem read
                        READ_SPRITE_PIXELS_BASE_WAIT: state_next= LOAD_SHIFT_REG;
```



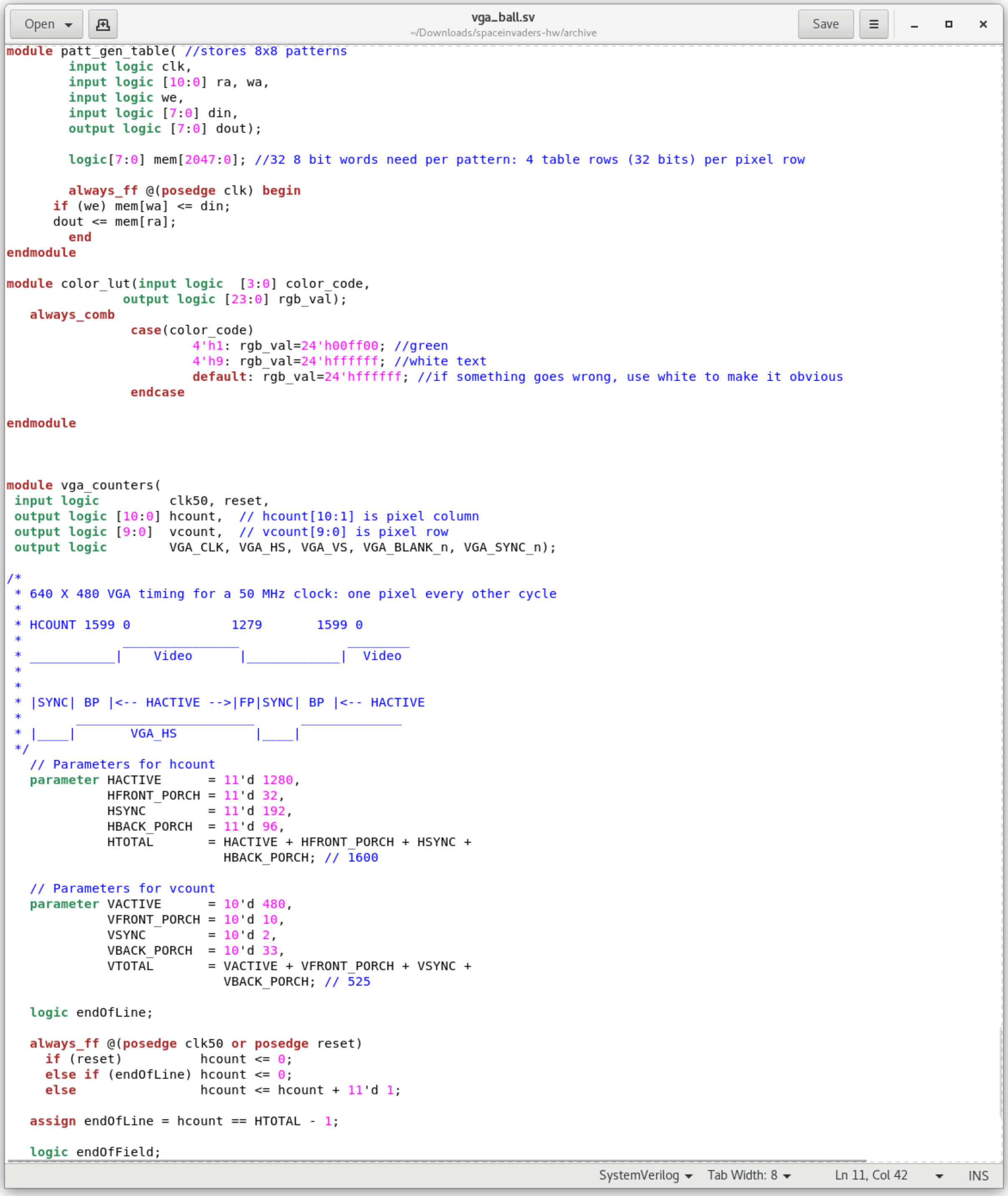
```
        always_comb begin
        case (state)
            IDLE:       state_next = (hcount==h_start) ? READ_VERT_POS: IDLE;
            READ_VERT_POS:      state_next = READ_VERT_POS_WAIT; //extra cycle for reading vertical position in attr table
                        READ_VERT_POS_WAIT:   state_next = READ_VERT_POS_WAIT2; //ra_a update needs 2 cycles for some reason
                        READ_VERT_POS_WAIT2: state_next = ((vcount [8:0]>={dout_a, 1'b0}) && (vcount[8:0]<({dout_a,
1'b0}+8'b10000)))? READ_HORT_POS: IDLE;  //check if any part of sprite is showing (don't need last 4 LSB)
                        READ_HORT_POS:       state_next = READ_HORT_POS_WAIT;  //extra cycle for mem read
                        READ_HORT_POS_WAIT:  state_next = READ_SPRITE_ADDR;
                        READ_SPRITE_ADDR:    state_next = READ_SPRITE_ADDR_WAIT; //extra cycle for mem read
                        READ_SPRITE_ADDR_WAIT:    state_next = READ_SPRITE_PIXELS_BASE;
                        READ_SPRITE_PIXELS_BASE: state_next= READ_SPRITE_PIXELS_BASE_WAIT; //extra cycle for mem read
                        READ_SPRITE_PIXELS_BASE_WAIT: state_next= LOAD_SHIFT_REG;
                        LOAD_SHIFT_REG: state_next= LOAD_SHIFT_REG_WAIT;
                        LOAD_SHIFT_REG_WAIT: state_next= (shift_reg_shift==8'b0) ? SPRITES_LOADED: LOAD_SHIFT_REG;

                        //if new vertical line started, begin down counting
                        SPRITES_LOADED: state_next= (hcount==11'b1111111) ? COUNT_DOWN : SPRITES_LOADED; //start at 127
                        COUNT_DOWN: state_next= (down_counter==9'b0) ? PREPARE_PIXELS: COUNT_DOWN;
                        PREPARE_PIXELS: state_next= (shift_pos==8'b0) ? IDLE : PREPARE_PIXELS;
            default:    state_next = IDLE;
        endcase
    end
endmodule

module pattern_prep (input logic clk, reset,
        input logic [10:0] hcount,
        input logic [9:0] vcount,
        input logic VGA_BLANK_n,
        input logic [7:0] dout_n,
        input logic [7:0] dout_g,
        output logic [11:0] ra_n,
        output logic [11:0] ra_g,
        output logic [3:0] out_pixel);

        logic [2047:0] shift_reg; //8*64*4 bit wide shift register
        logic [11:0] shift_pos; //position in shift reg to read pixel value from
        logic [10:0] pattern_row_offset; //which of 8 of a given pattern to display
        logic [2047:0] display_pixel;// determines whether sprite or background pixel is shown
        logic [11:0] shift_reg_shift; //bit position in shift reg to write to (0-63, steps of 8)
        logic [7:0] tile_total_counter; //counts the total number of tiles that has been loaded into shift reg
        logic [7:0] tile_pixel_counter; //counts the number of tile pixel rows that has been loaded
        assign out_pixel=display_pixel[3:0];

        parameter [11:0] v_start=12'h0; //vertical position where first pattern begins
        parameter [7:0] tiles_per_row=8'd64; //number of tiles per row
        parameter [11:0] name_table_addr_mask={6'b111111, 6'b0};

        enum {IDLE, READ_TILE_ADDR_BASE, READ_TILE_ADDR_BASE_WAIT, READ_PATT_PIXELS_BASE, READ_PATT_PIXELS_BASE_WAIT,
        LOAD_SHIFT_REG, LOAD_SHIFT_REG_WAIT, READ_TILE_NEXT, READ_TILE_NEXT_WAIT, PATT_LOADED, PREPARE_PIXELS }
        state, state_next;


        always_ff @(posedge clk) begin
                state<=state_next;
                if (reset) begin
                        state<=IDLE;
                        ra_n<=0;
                        ra_g<=0;
                end

                case (state)
                        IDLE: begin
                                tile_total_counter<=8'b0;
                                tile_pixel_counter<=8'b0;
                                display_pixel<=2048'b0;
                                shift_reg<=2048'b0;
                                shift_reg_shift<=12'b100000000000; //dec=2048 (actual value used is 8 less)
                                shift_pos<=12'b100000000000; // dec=2048 set shift position to start of shift regs (MSB)
(actual value used is 4 less)

                        end
```



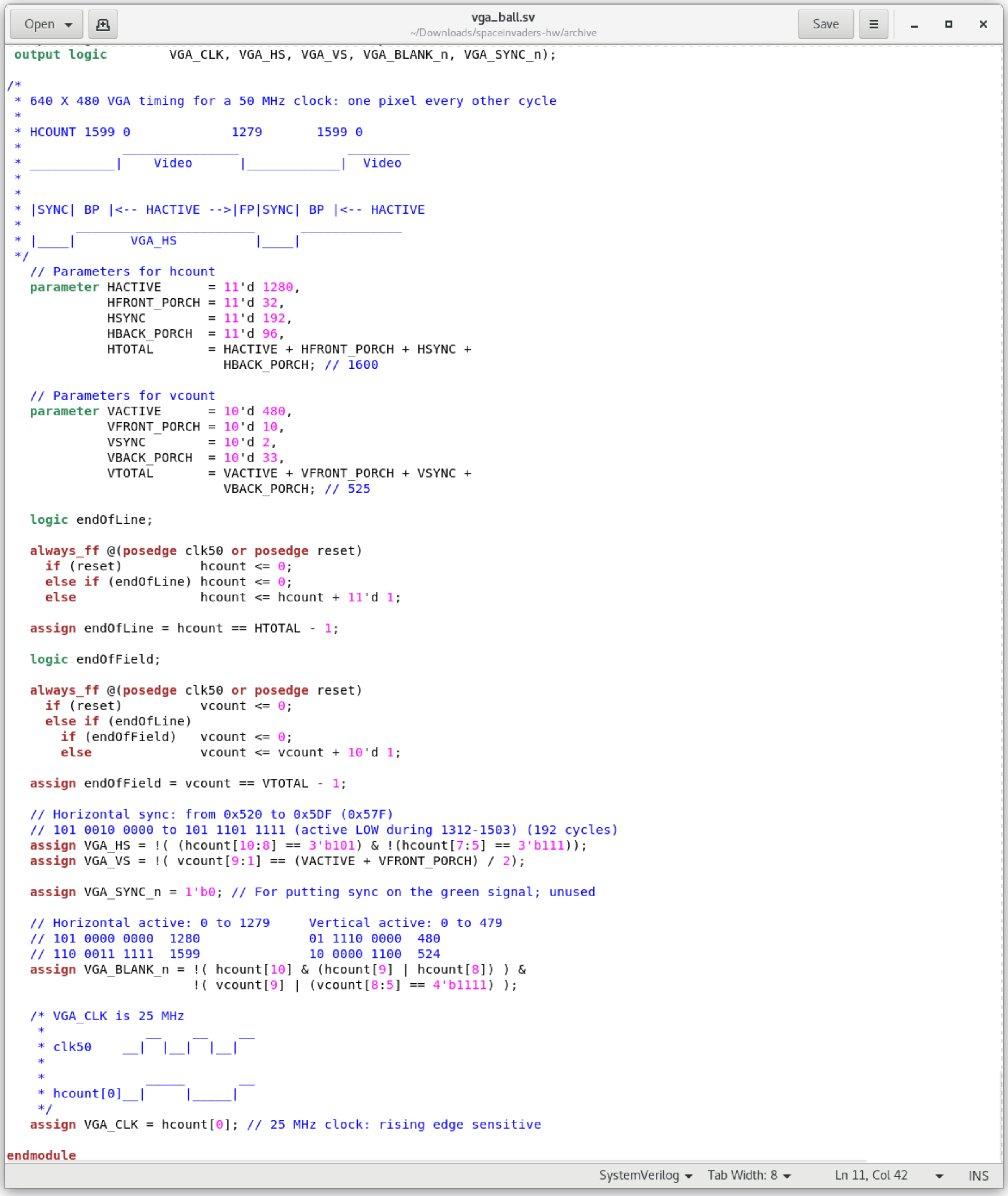
```
                                shift_reg<=2048'b0;
                                shift_reg_shift<=12'b100000000000; //dec=2048 (actual value used is 8 less)
                                shift_pos<=12'b100000000000; // dec=2048 set shift position to start of shift regs (MSB)
(actual value used is 4 less)

                        end
                        READ_TILE_ADDR_BASE: begin
                                ra_n<=(({2'b0, vcount}-v_start)<<3) & name_table_addr_mask; //get address of (starting) tile
pixel address in name table
                                pattern_row_offset<={8'b0, vcount[2:0]-v_start[2:0]}; //which of 8 pixel rows to access
                        end

                        READ_PATT_PIXELS_BASE: begin //!!note: address no longer >> shifted by 3!!
                                ra_g<={dout_n[5:0], 5'b0} + (pattern_row_offset<<2); //read base 8 pixels in gen table,4x
offset since 4 table rows needed per pixel line
                        end

                        READ_PATT_PIXELS_BASE_WAIT: begin //!!note: address no longer >> shifted by 3!!
                                ra_g<=ra_g+1;
                        end

                        LOAD_SHIFT_REG: begin //first time: gets ra_g pixels_base stage and not base_wait stage
                                shift_reg<= ({2040'b0, dout_g}<<(shift_reg_shift-12'h8)) | shift_reg; //store left-most 8
pixels of sprite line
                                shift_reg_shift<=shift_reg_shift-12'h8; //minus 8
                                ra_g<=ra_g+1; //increment gen table address by one to read upcoming pixels
                                tile_pixel_counter<=tile_pixel_counter+8'b1;
                        end
                        READ_TILE_NEXT: begin
                                ra_n<=ra_n+1; //increment name table address
                                tile_pixel_counter<=8'b0;
                                tile_total_counter<=tile_total_counter+8'b1;
                        end

                        PREPARE_PIXELS: begin
                                if (VGA_BLANK_n && !hcount[0]) begin
                                        display_pixel<=(shift_reg>>(shift_pos-12'h4)); //Only 4 LSB of display_pixel matter
                                        shift_pos<=shift_pos-12'h4; //minus 4
                                end
                        end
                endcase

        end

        always_comb begin
        case (state)
            IDLE:       state_next = ((hcount==11'd1152) && (vcount>=v_start[9:0]) && (vcount<10'd480)) ? READ_TILE_ADDR_BASE:
IDLE; //start at h=1152 and vcount=0
            READ_TILE_ADDR_BASE:      state_next = READ_TILE_ADDR_BASE_WAIT; //extra cycle for reading vertical position in
attr table
                        READ_TILE_ADDR_BASE_WAIT:   state_next = READ_PATT_PIXELS_BASE; //check if true: ra_a update needs 2
cycles for some reason
                        READ_PATT_PIXELS_BASE:          state_next = READ_PATT_PIXELS_BASE_WAIT;
                        READ_PATT_PIXELS_BASE_WAIT: state_next= LOAD_SHIFT_REG;
                        LOAD_SHIFT_REG: state_next= (tile_pixel_counter==8'h3) ? READ_TILE_NEXT: LOAD_SHIFT_REG;
                        READ_TILE_NEXT: state_next=READ_TILE_NEXT_WAIT;
                        READ_TILE_NEXT_WAIT: state_next=(tile_total_counter==tiles_per_row)? PATT_LOADED: READ_PATT_PIXELS_BASE;

                        //if new vertical line started, begin down counting
                        PATT_LOADED: state_next= (hcount==11'd127) ? PREPARE_PIXELS : PATT_LOADED;
                        PREPARE_PIXELS: state_next= (shift_pos==12'b0 || vcount>10'd480) ? IDLE : PREPARE_PIXELS;
            default:    state_next = IDLE;
        endcase
    end
endmodule

module sprite_attr_table( //stores sprite information (x, y, name, color)
        //x and y position has to be a multiple (2x) of hcount/vcount since only 8 bits
        input logic clk,
        input logic [31:0] ra, wa, //change later
        input logic we,
        input logic [7:0] din,
```



```
            IDLE:       state_next = ((hcount==11'd1152) && (vcount>=v_start[9:0]) && (vcount<10'd480)) ? READ_TILE_ADDR_BASE:
IDLE; //start at h=1152 and vcount=0
            READ_TILE_ADDR_BASE:      state_next = READ_TILE_ADDR_BASE_WAIT; //extra cycle for reading vertical position in
attr table
                        READ_TILE_ADDR_BASE_WAIT:   state_next = READ_PATT_PIXELS_BASE; //check if true: ra_a update needs 2
cycles for some reason
                        READ_PATT_PIXELS_BASE:          state_next = READ_PATT_PIXELS_BASE_WAIT;
                        READ_PATT_PIXELS_BASE_WAIT: state_next= LOAD_SHIFT_REG;
                        LOAD_SHIFT_REG: state_next= (tile_pixel_counter==8'h3) ? READ_TILE_NEXT: LOAD_SHIFT_REG;
                        READ_TILE_NEXT: state_next=READ_TILE_NEXT_WAIT;
                        READ_TILE_NEXT_WAIT: state_next=(tile_total_counter==tiles_per_row)? PATT_LOADED: READ_PATT_PIXELS_BASE;

                        //if new vertical line started, begin down counting
                        PATT_LOADED: state_next= (hcount==11'd127) ? PREPARE_PIXELS : PATT_LOADED;
                        PREPARE_PIXELS: state_next= (shift_pos==12'b0 || vcount>10'd480) ? IDLE : PREPARE_PIXELS;
            default:    state_next = IDLE;
        endcase
    end
endmodule

module sprite_attr_table( //stores sprite information (x, y, name, color)
        //x and y position has to be a multiple (2x) of hcount/vcount since only 8 bits
        input logic clk,
        input logic [31:0] ra, wa, //change later
        input logic we,
        input logic [7:0] din,
        output logic [7:0] dout);

        logic[7:0] mem[31:0];

        always_ff @(posedge clk) begin
      if (we) mem[wa] <= din;
      dout <= mem[ra];
        end
endmodule

module sprite_gen_table(
        input logic clk,
        input logic [11:0] ra, wa, //change later
        input logic we,
        input logic [7:0] din,
        output logic [7:0] dout);

        logic[7:0] mem[4095:0]; //128 8 bit words need per sprite:

        always_ff @(posedge clk) begin
      if (we) mem[wa] <= din;
      dout <= mem[ra];
        end
endmodule

module patt_name_table( //stores 8 bit address of tiles on each row
        input logic clk,
        input logic [11:0] ra, wa, //12 bit addr
        input logic we,
        input logic [7:0] din,
        output logic [7:0] dout);

        logic[7:0] mem[4095:0];

        always_ff @(posedge clk) begin
      if (we) mem[wa] <= din;
      dout <= mem[ra];
        end
endmodule

module patt_gen_table( //stores 8x8 patterns
        input logic clk,
        input logic [10:0] ra, wa,
        input logic we,
        input logic [7:0] din,
        output logic [7:0] dout);
```



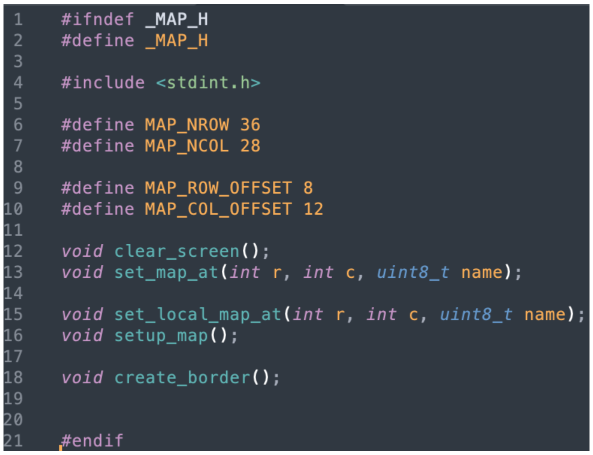
```
module patt_gen_table( //stores 8x8 patterns
        input logic clk,
        input logic [10:0] ra, wa,
        input logic we,
        input logic [7:0] din,
        output logic [7:0] dout);

        logic[7:0] mem[2047:0]; //32 8 bit words need per pattern: 4 table rows (32 bits) per pixel row

        always_ff @(posedge clk) begin
      if (we) mem[wa] <= din;
      dout <= mem[ra];
        end
endmodule

module color_lut(input logic  [3:0] color_code,
               output logic [23:0] rgb_val);
   always_comb
                case(color_code)
                        4'h1: rgb_val=24'h00ff00; //green
                        4'h9: rgb_val=24'hffffff; //white text
                        default: rgb_val=24'hffffff; //if something goes wrong, use white to make it obvious
                endcase

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;
```



```
 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 (active LOW during 1312-1503) (192 cycles)
   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
```

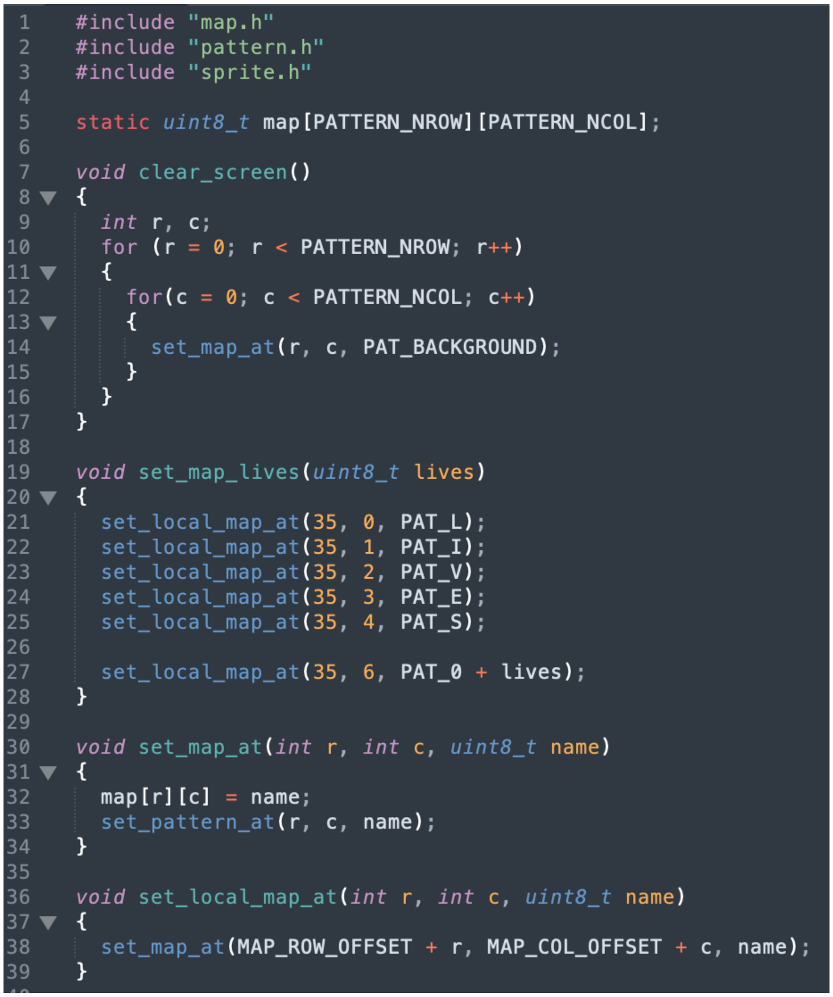
Code Screenshots - Software

map.h



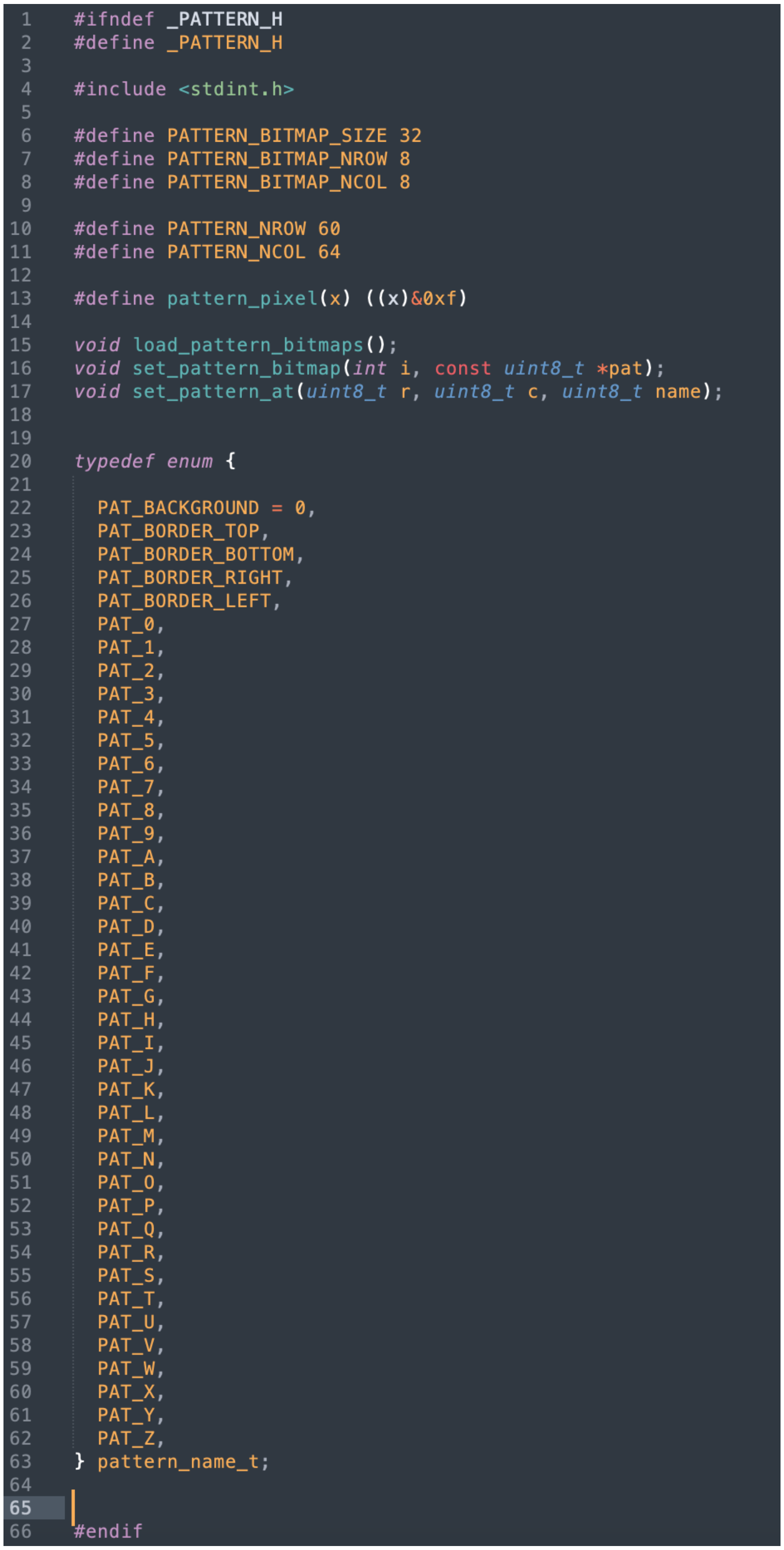
```c
#ifndef _MAP_H
#define _MAP_H

#include <stdint.h>

#define MAP_NROW 36
#define MAP_NCOL 28

#define MAP_ROW_OFFSET 8
#define MAP_COL_OFFSET 12

void clear_screen();
void set_map_at(int r, int c, uint8_t name);

void set_local_map_at(int r, int c, uint8_t name);
void setup_map();

void create_border();


#endif
```

map.c



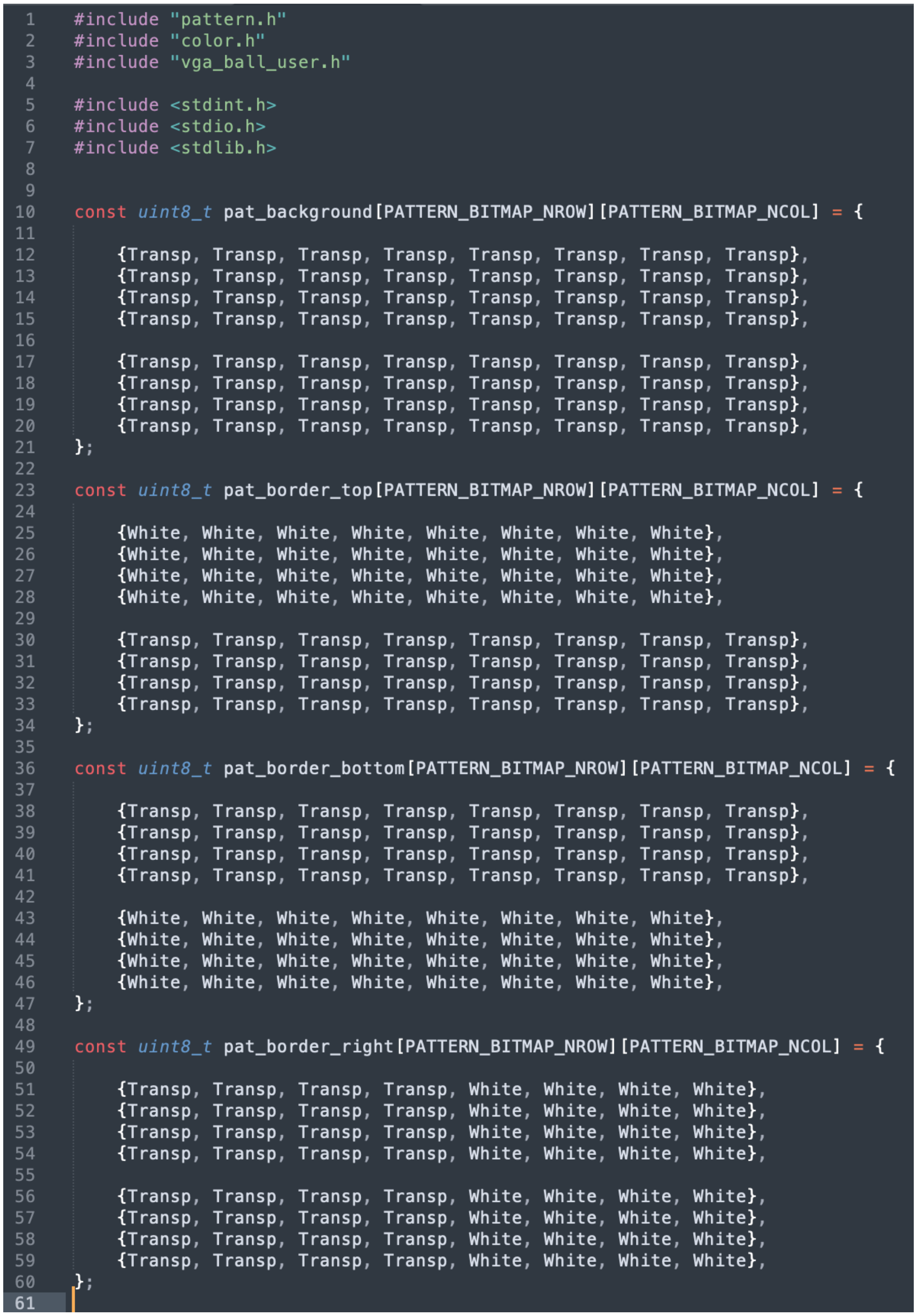
```c
#include "map.h"
#include "pattern.h"
#include "sprite.h"

static uint8_t map[PATTERN_NROW][PATTERN_NCOL];

void clear_screen()
{
  int r, c;
  for (r = 0; r < PATTERN_NROW; r++)
  {
    for(c = 0; c < PATTERN_NCOL; c++)
    {
      set_map_at(r, c, PAT_BACKGROUND);
    }
  }
}

void set_map_lives(uint8_t lives)
{
  set_local_map_at(35, 0, PAT_L);
  set_local_map_at(35, 1, PAT_I);
  set_local_map_at(35, 2, PAT_V);
  set_local_map_at(35, 3, PAT_E);
  set_local_map_at(35, 4, PAT_S);

  set_local_map_at(35, 6, PAT_0 + lives);
}

void set_map_at(int r, int c, uint8_t name)
{
  map[r][c] = name;
  set_pattern_at(r, c, name);
}

void set_local_map_at(int r, int c, uint8_t name)
{
  set_map_at(MAP_ROW_OFFSET + r, MAP_COL_OFFSET + c, name);
}
```

pattern.h

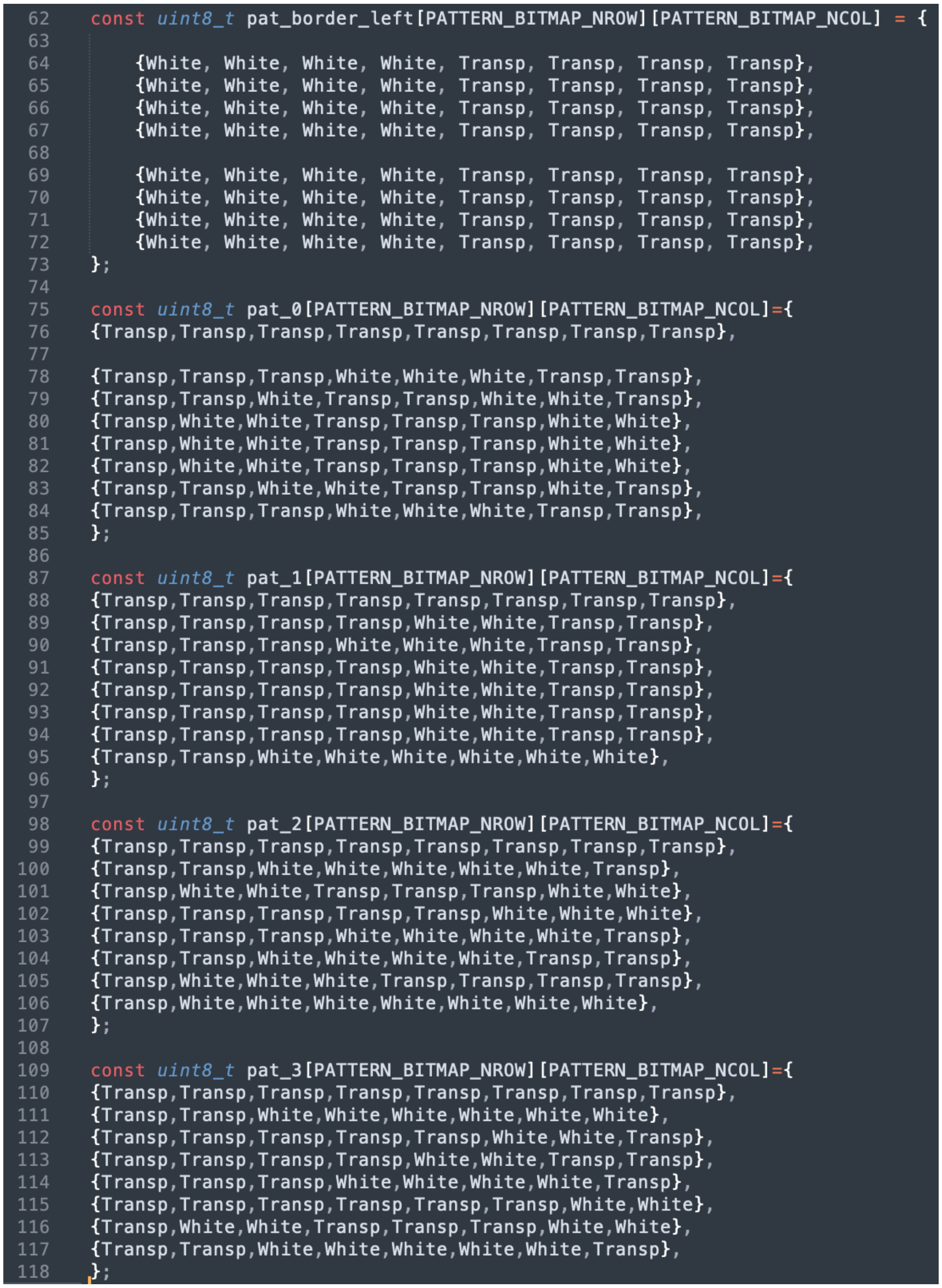


```c
#ifndef _PATTERN_H
#define _PATTERN_H

#include <stdint.h>

#define PATTERN_BITMAP_SIZE 32
#define PATTERN_BITMAP_NROW 8
#define PATTERN_BITMAP_NCOL 8

#define PATTERN_NROW 60
#define PATTERN_NCOL 64

#define pattern_pixel(x) ((x)&0xf)

void load_pattern_bitmaps();
void set_pattern_bitmap(int i, const uint8_t *pat);
void set_pattern_at(uint8_t r, uint8_t c, uint8_t name);


typedef enum {

  PAT_BACKGROUND = 0,
  PAT_BORDER_TOP,
  PAT_BORDER_BOTTOM,
  PAT_BORDER_RIGHT,
  PAT_BORDER_LEFT,
  PAT_0,
  PAT_1,
  PAT_2,
  PAT_3,
  PAT_4,
  PAT_5,
  PAT_6,
  PAT_7,
  PAT_8,
  PAT_9,
  PAT_A,
  PAT_B,
  PAT_C,
  PAT_D,
  PAT_E,
  PAT_F,
  PAT_G,
  PAT_H,
  PAT_I,
  PAT_J,
  PAT_K,
  PAT_L,
  PAT_M,
  PAT_N,
  PAT_O,
  PAT_P,
  PAT_Q,
  PAT_R,
  PAT_S,
  PAT_T,
  PAT_U,
  PAT_V,
  PAT_W,
  PAT_X,
  PAT_Y,
  PAT_Z,
} pattern_name_t;


#endif
```

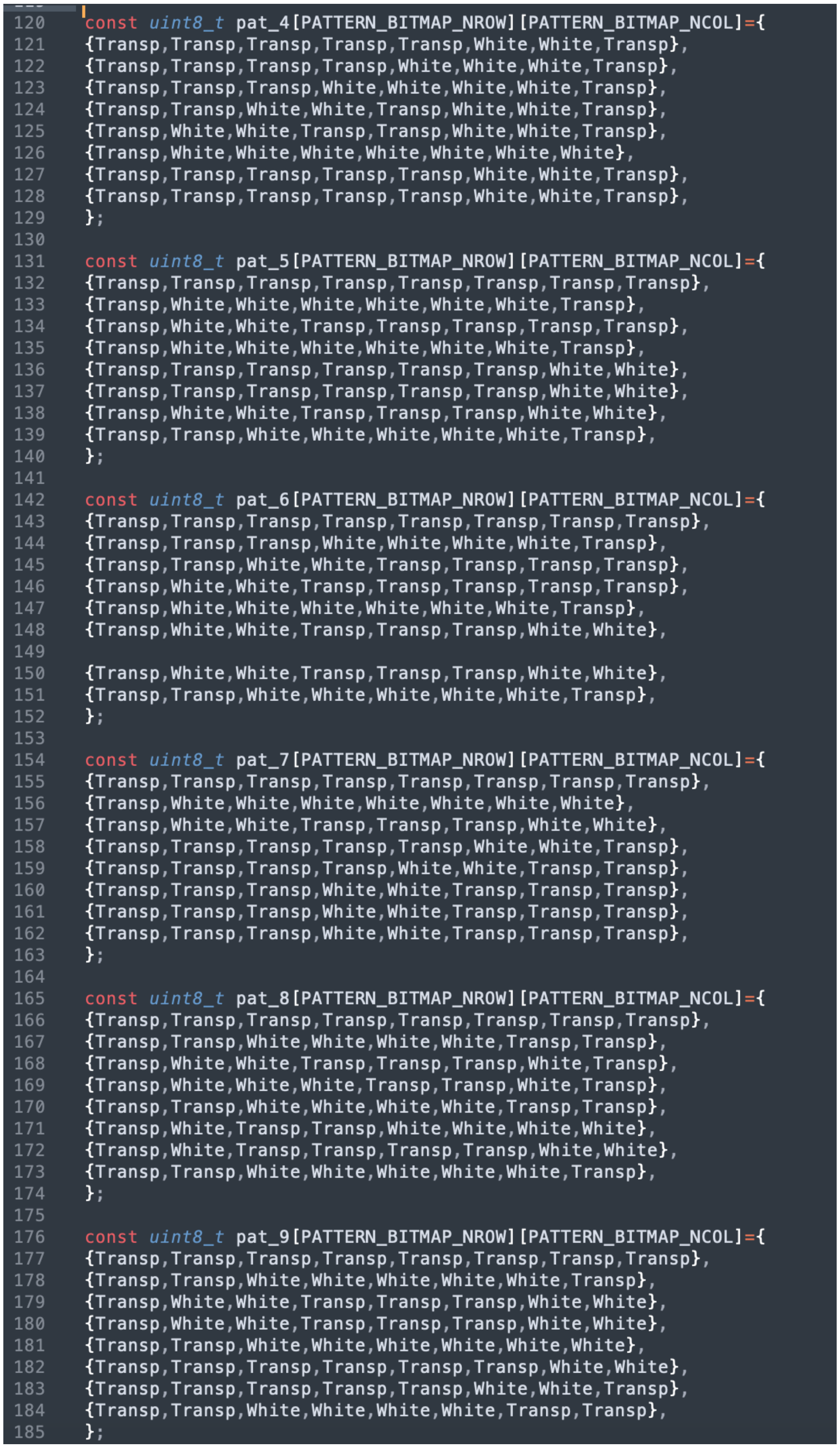
pattern.c



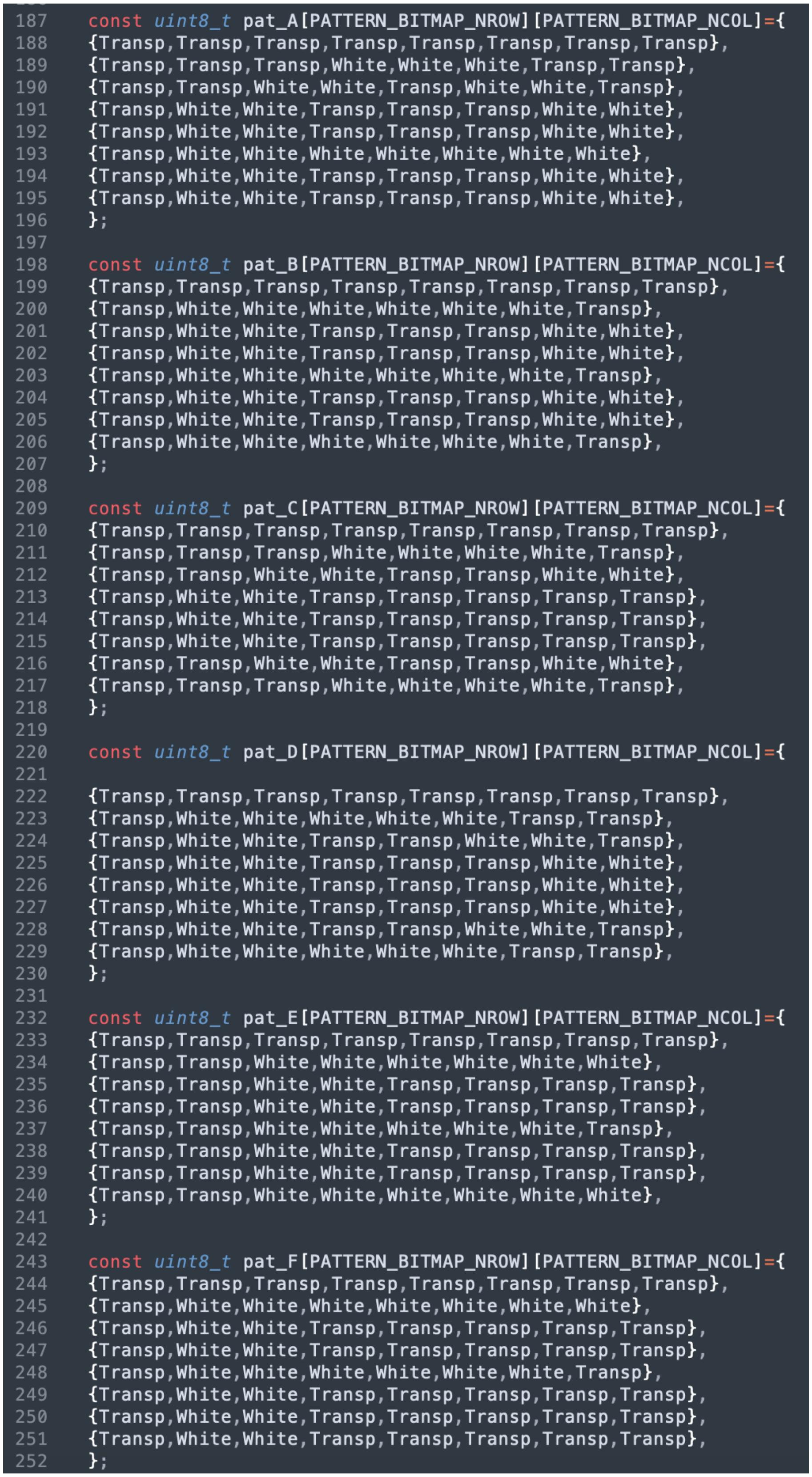
```c
#include "pattern.h"
#include "color.h"
#include "vga_ball_user.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>


const uint8_t pat_background[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL] = {

    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
};

const uint8_t pat_border_top[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL] = {

    {White, White, White, White, White, White, White, White},
    {White, White, White, White, White, White, White, White},
    {White, White, White, White, White, White, White, White},
    {White, White, White, White, White, White, White, White},

    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
};

const uint8_t pat_border_bottom[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL] = {

    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
    {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

    {White, White, White, White, White, White, White, White},
    {White, White, White, White, White, White, White, White},
    {White, White, White, White, White, White, White, White},
    {White, White, White, White, White, White, White, White},
};

const uint8_t pat_border_right[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL] = {

    {Transp, Transp, Transp, Transp, White, White, White, White},
    {Transp, Transp, Transp, Transp, White, White, White, White},
    {Transp, Transp, Transp, Transp, White, White, White, White},
    {Transp, Transp, Transp, Transp, White, White, White, White},

    {Transp, Transp, Transp, Transp, White, White, White, White},
    {Transp, Transp, Transp, Transp, White, White, White, White},
    {Transp, Transp, Transp, Transp, White, White, White, White},
    {Transp, Transp, Transp, Transp, White, White, White, White},
};

```



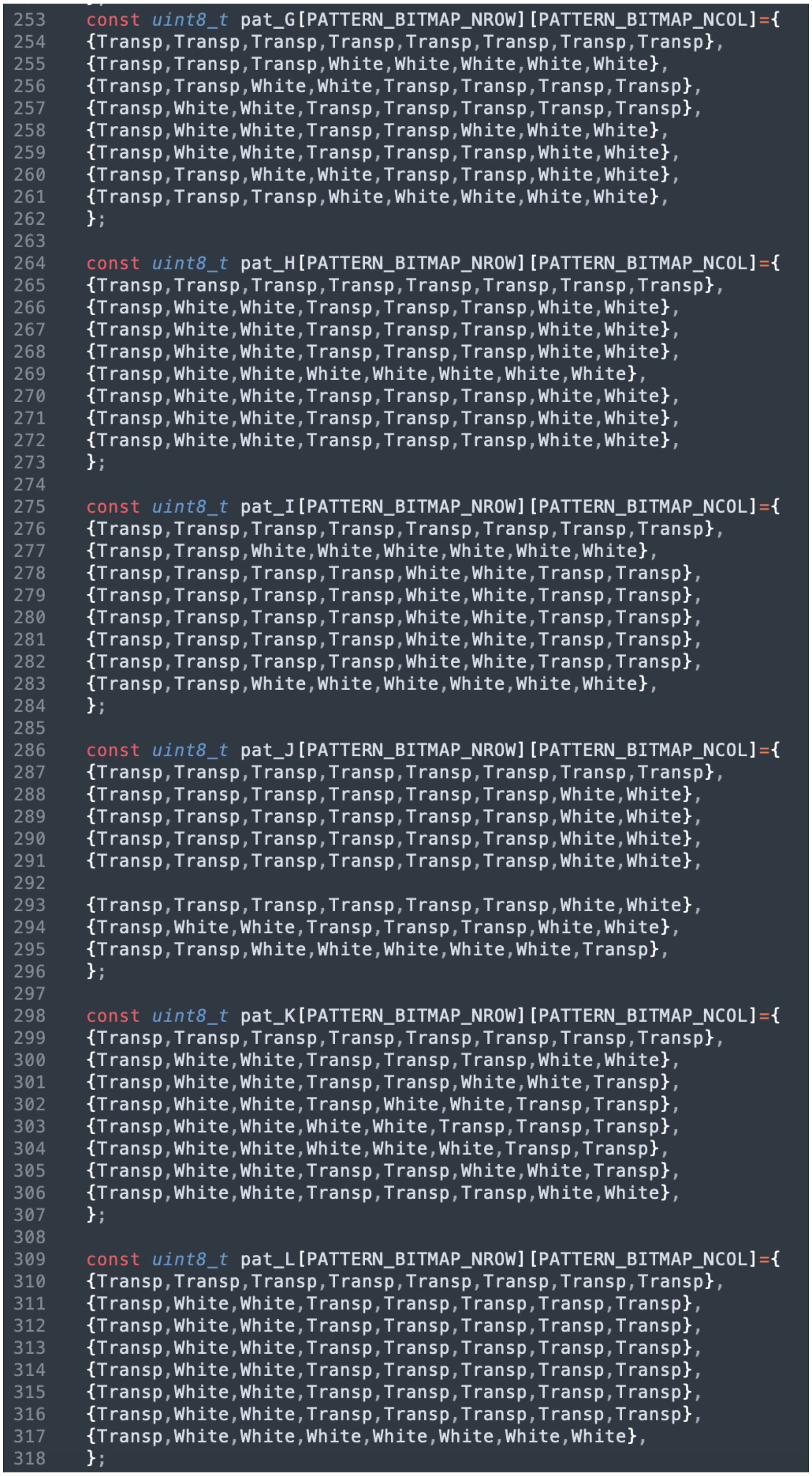
```
const uint8_t pat_border_left[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL] = {

    {White, White, White, White, Transp, Transp, Transp, Transp},
    {White, White, White, White, Transp, Transp, Transp, Transp},
    {White, White, White, White, Transp, Transp, Transp, Transp},
    {White, White, White, White, Transp, Transp, Transp, Transp},

    {White, White, White, White, Transp, Transp, Transp, Transp},
    {White, White, White, White, Transp, Transp, Transp, Transp},
    {White, White, White, White, Transp, Transp, Transp, Transp},
    {White, White, White, White, Transp, Transp, Transp, Transp},
};

const uint8_t pat_0[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},

{Transp,Transp,Transp,White,White,White,Transp,Transp},
{Transp,Transp,White,Transp,Transp,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,Transp,Transp,White,Transp},
{Transp,Transp,Transp,White,White,White,Transp,Transp},
};

const uint8_t pat_1[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,White,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
};

const uint8_t pat_2[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,White,White,White},
{Transp,Transp,Transp,White,White,White,White,Transp},
{Transp,Transp,White,White,White,White,Transp,Transp},
{Transp,White,White,White,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,White},
};

const uint8_t pat_3[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
{Transp,Transp,Transp,Transp,Transp,White,White,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,White,White,White,White,Transp},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};
```



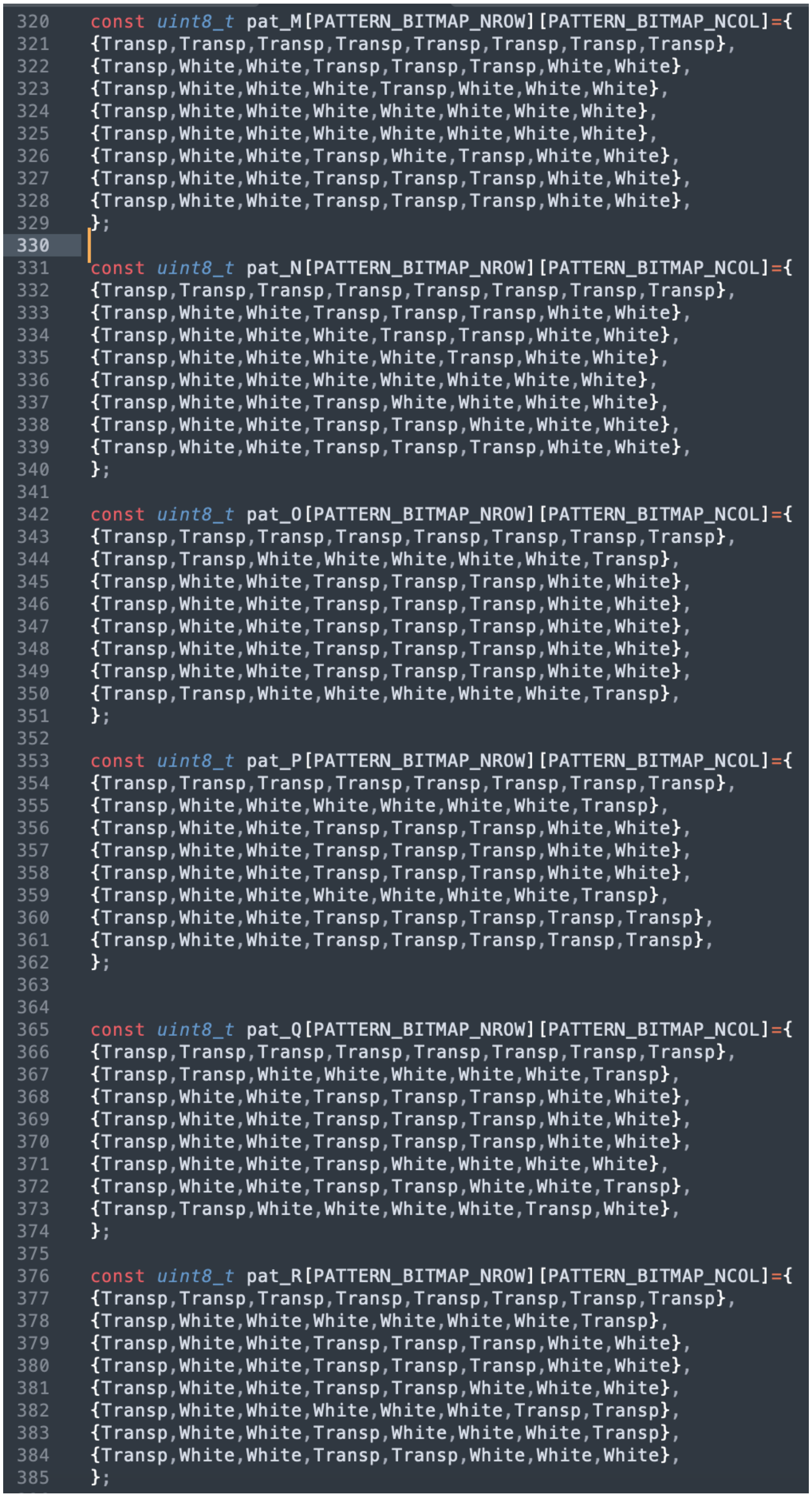
```c
const uint8_t pat_4[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,White,White,Transp},
{Transp,Transp,Transp,Transp,White,White,White,Transp},
{Transp,Transp,Transp,White,White,White,White,Transp},
{Transp,Transp,White,White,Transp,White,White,Transp},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,White,White,White,White,White,White,White},
{Transp,Transp,Transp,Transp,Transp,White,White,Transp},
{Transp,Transp,Transp,Transp,Transp,White,White,Transp},
};

const uint8_t pat_5[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_6[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,Transp,White,White,White,White,Transp},
{Transp,Transp,White,White,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},

{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_7[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,White,White,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,White,White,Transp,Transp,Transp},
{Transp,Transp,Transp,White,White,Transp,Transp,Transp},
{Transp,Transp,Transp,White,White,Transp,Transp,Transp},
};

const uint8_t pat_8[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,Transp},
{Transp,White,White,White,Transp,Transp,White,Transp},
{Transp,Transp,White,White,White,White,Transp,Transp},
{Transp,White,Transp,Transp,White,White,White,White},
{Transp,White,Transp,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_9[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,White},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,White,White,Transp},
{Transp,Transp,White,White,White,White,Transp,Transp},
};
```



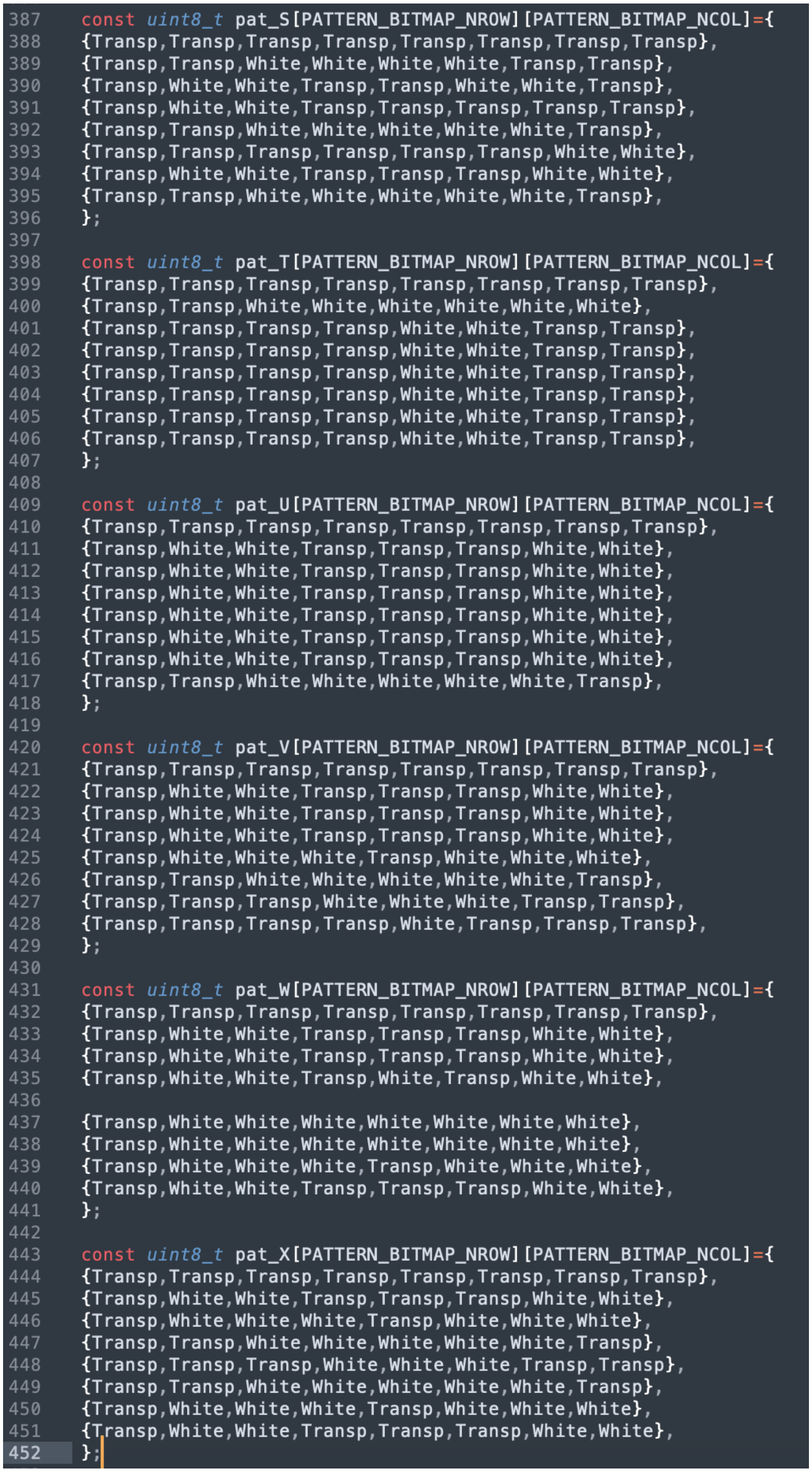
```
const uint8_t pat_A[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,Transp,White,White,White,Transp,Transp},
{Transp,Transp,White,White,Transp,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
};

const uint8_t pat_B[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,White,White,White,Transp},
};

const uint8_t pat_C[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,Transp,White,White,White,White,Transp},
{Transp,Transp,White,White,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,Transp,Transp,White,White},
{Transp,Transp,Transp,White,White,White,White,Transp},
};

const uint8_t pat_D[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={

{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,Transp,Transp},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,White,White,White,White,White,Transp,Transp},
};

const uint8_t pat_E[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
{Transp,Transp,White,White,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,Transp,White,White,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
};

const uint8_t pat_F[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
};
```



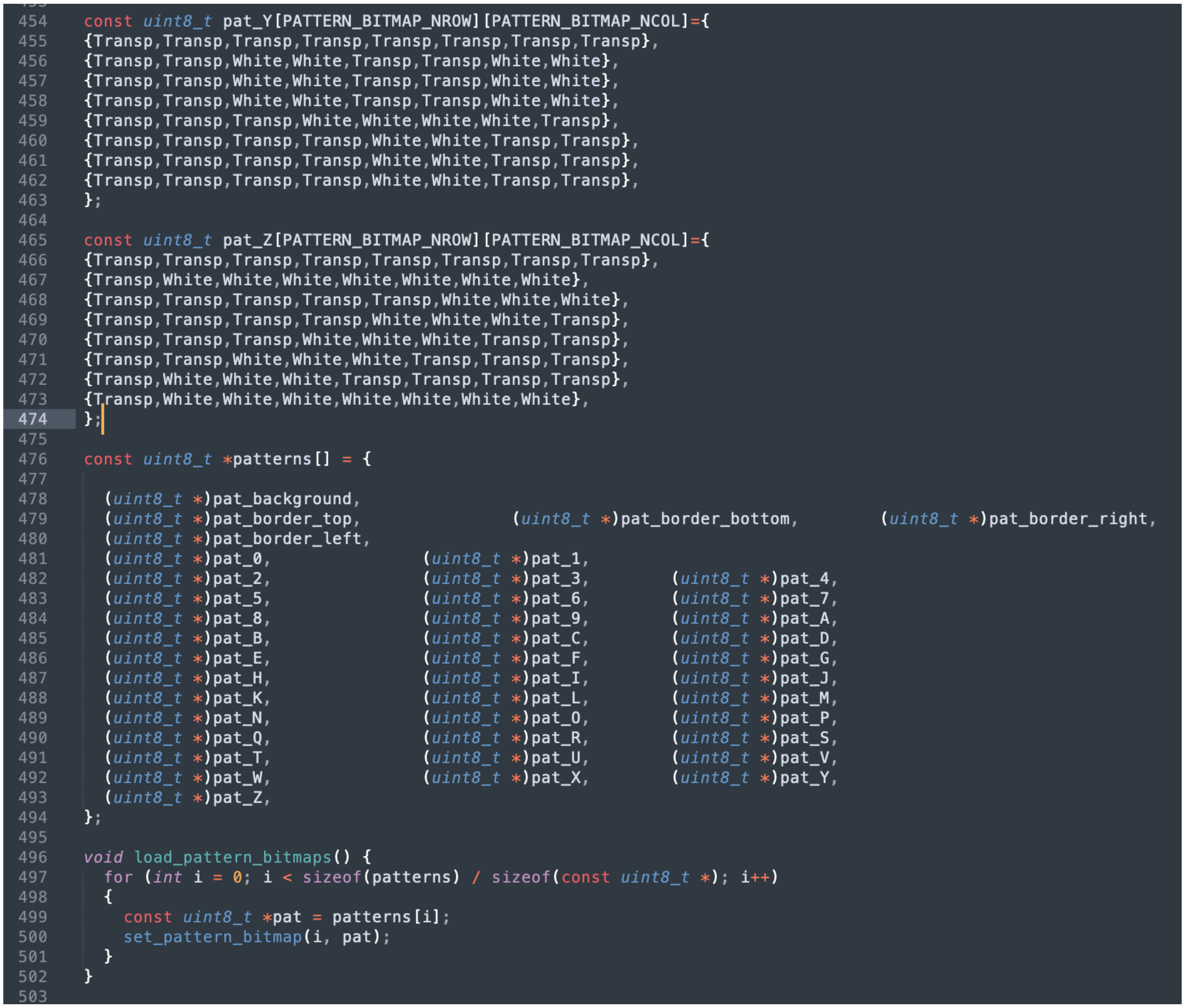
```
const uint8_t pat_G[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,Transp,White,White,White,White,White},
{Transp,Transp,White,White,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,Transp,Transp,White,White},
{Transp,Transp,Transp,White,White,White,White,White},
};

const uint8_t pat_H[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
};

const uint8_t pat_I[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
};

const uint8_t pat_J[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},

{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_K[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,White,White,Transp,White,White,Transp,Transp},
{Transp,White,White,White,White,Transp,Transp,Transp},
{Transp,White,White,White,White,White,Transp,Transp},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
};

const uint8_t pat_L[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,White},
};
```



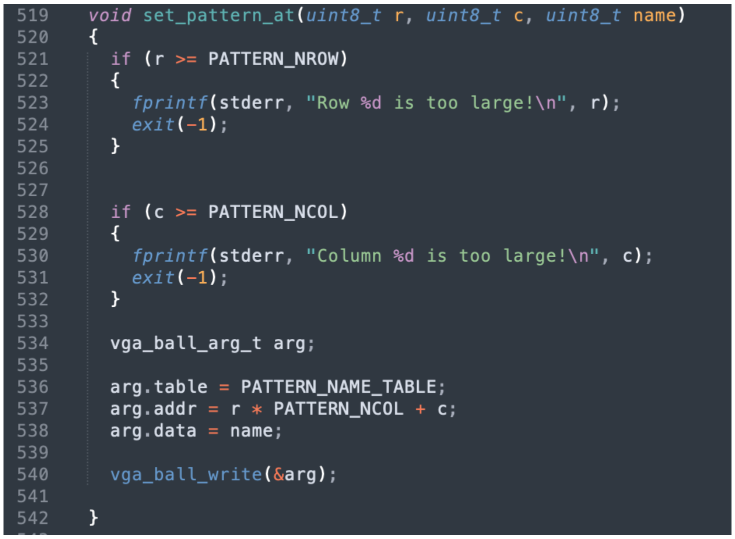
```
const uint8_t pat_M[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,Transp,White,White,White},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,Transp,White,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
};

const uint8_t pat_N[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,Transp,Transp,White,White},
{Transp,White,White,White,White,Transp,White,White},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,Transp,White,White,White,White},
{Transp,White,White,Transp,Transp,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
};

const uint8_t pat_O[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_P[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
};


const uint8_t pat_Q[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,White,White,White,White},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,Transp,White,White,White,White,Transp,White},
};

const uint8_t pat_R[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,White,White,White},
{Transp,White,White,White,White,White,Transp,Transp},
{Transp,White,White,Transp,White,White,White,Transp},
{Transp,White,White,Transp,Transp,White,White,White},
};
```



```
const uint8_t pat_S[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,Transp,Transp},
{Transp,White,White,Transp,Transp,White,White,Transp},
{Transp,White,White,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,Transp,Transp,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_T[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,White,White,White,White},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
};

const uint8_t pat_U[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
};

const uint8_t pat_V[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,Transp,White,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,Transp,Transp,White,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,Transp,Transp,Transp},
};

const uint8_t pat_W[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,Transp,White,Transp,White,White},

{Transp,White,White,White,White,White,White,White},
{Transp,White,White,White,White,White,White,White},
{Transp,White,White,White,Transp,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
};

const uint8_t pat_X[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,Transp,Transp,Transp,White,White},
{Transp,White,White,White,Transp,White,White,White},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,Transp,Transp,White,White,White,Transp,Transp},
{Transp,Transp,White,White,White,White,White,Transp},
{Transp,White,White,White,Transp,White,White,White},
{Transp,White,White,Transp,Transp,Transp,White,White},
};
```

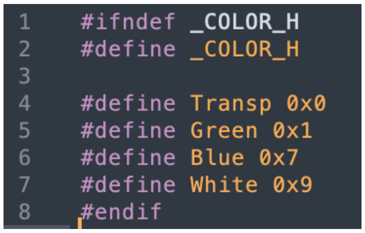


```
const uint8_t pat_Y[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,Transp,White,White,Transp,Transp,White,White},
{Transp,Transp,White,White,Transp,Transp,White,White},
{Transp,Transp,White,White,Transp,Transp,White,White},
{Transp,Transp,Transp,White,White,White,White,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
{Transp,Transp,Transp,Transp,White,White,Transp,Transp},
};

const uint8_t pat_Z[PATTERN_BITMAP_NROW][PATTERN_BITMAP_NCOL]={
{Transp,Transp,Transp,Transp,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,White},
{Transp,Transp,Transp,Transp,Transp,White,White,White},
{Transp,Transp,Transp,Transp,White,White,White,Transp},
{Transp,Transp,Transp,White,White,White,Transp,Transp},
{Transp,Transp,White,White,White,Transp,Transp,Transp},
{Transp,White,White,White,Transp,Transp,Transp,Transp},
{Transp,White,White,White,White,White,White,White},
};

const uint8_t *patterns[] = {

  (uint8_t *)pat_background,
  (uint8_t *)pat_border_top,               (uint8_t *)pat_border_bottom,         (uint8_t *)pat_border_right,
  (uint8_t *)pat_border_left,
  (uint8_t *)pat_0,               (uint8_t *)pat_1,
  (uint8_t *)pat_2,               (uint8_t *)pat_3,         (uint8_t *)pat_4,
  (uint8_t *)pat_5,               (uint8_t *)pat_6,         (uint8_t *)pat_7,
  (uint8_t *)pat_8,               (uint8_t *)pat_9,         (uint8_t *)pat_A,
  (uint8_t *)pat_B,               (uint8_t *)pat_C,         (uint8_t *)pat_D,
  (uint8_t *)pat_E,               (uint8_t *)pat_F,         (uint8_t *)pat_G,
  (uint8_t *)pat_H,               (uint8_t *)pat_I,         (uint8_t *)pat_J,
  (uint8_t *)pat_K,               (uint8_t *)pat_L,         (uint8_t *)pat_M,
  (uint8_t *)pat_N,               (uint8_t *)pat_O,         (uint8_t *)pat_P,
  (uint8_t *)pat_Q,               (uint8_t *)pat_R,         (uint8_t *)pat_S,
  (uint8_t *)pat_T,               (uint8_t *)pat_U,         (uint8_t *)pat_V,
  (uint8_t *)pat_W,               (uint8_t *)pat_X,         (uint8_t *)pat_Y,
  (uint8_t *)pat_Z,
};

void load_pattern_bitmaps() {
  for (int i = 0; i < sizeof(patterns) / sizeof(const uint8_t *); i++)
  {
    const uint8_t *pat = patterns[i];
    set_pattern_bitmap(i, pat);
  }
}
```



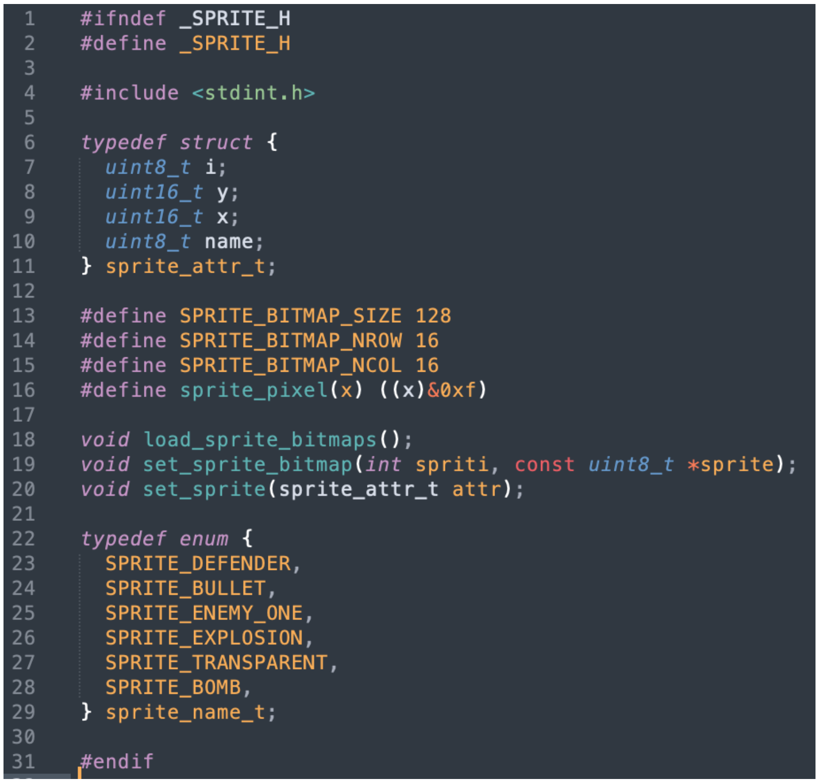
```
void set_pattern_at(uint8_t r, uint8_t c, uint8_t name)
{
  if (r >= PATTERN_NROW)
  {
    fprintf(stderr, "Row %d is too large!\n", r);
    exit(-1);
  }


  if (c >= PATTERN_NCOL)
  {
    fprintf(stderr, "Column %d is too large!\n", c);
    exit(-1);
  }

  vga_ball_arg_t arg;

  arg.table = PATTERN_NAME_TABLE;
  arg.addr = r * PATTERN_NCOL + c;
  arg.data = name;

  vga_ball_write(&arg);

}
```

color.h



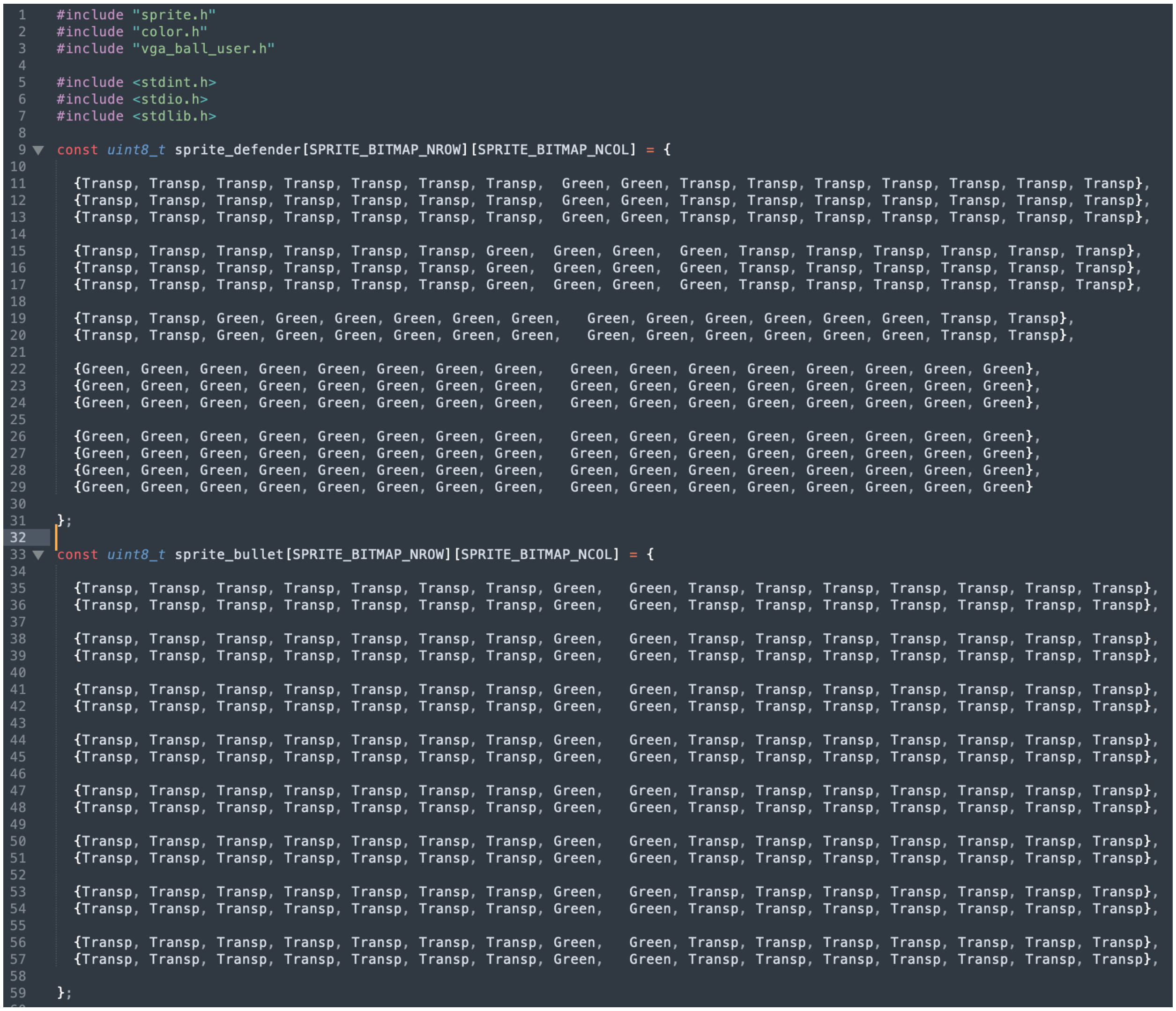
```c
#ifndef _COLOR_H
#define _COLOR_H

#define Transp 0x0
#define Green 0x1
#define Blue 0x7
#define White 0x9
#endif
```

sprite.h

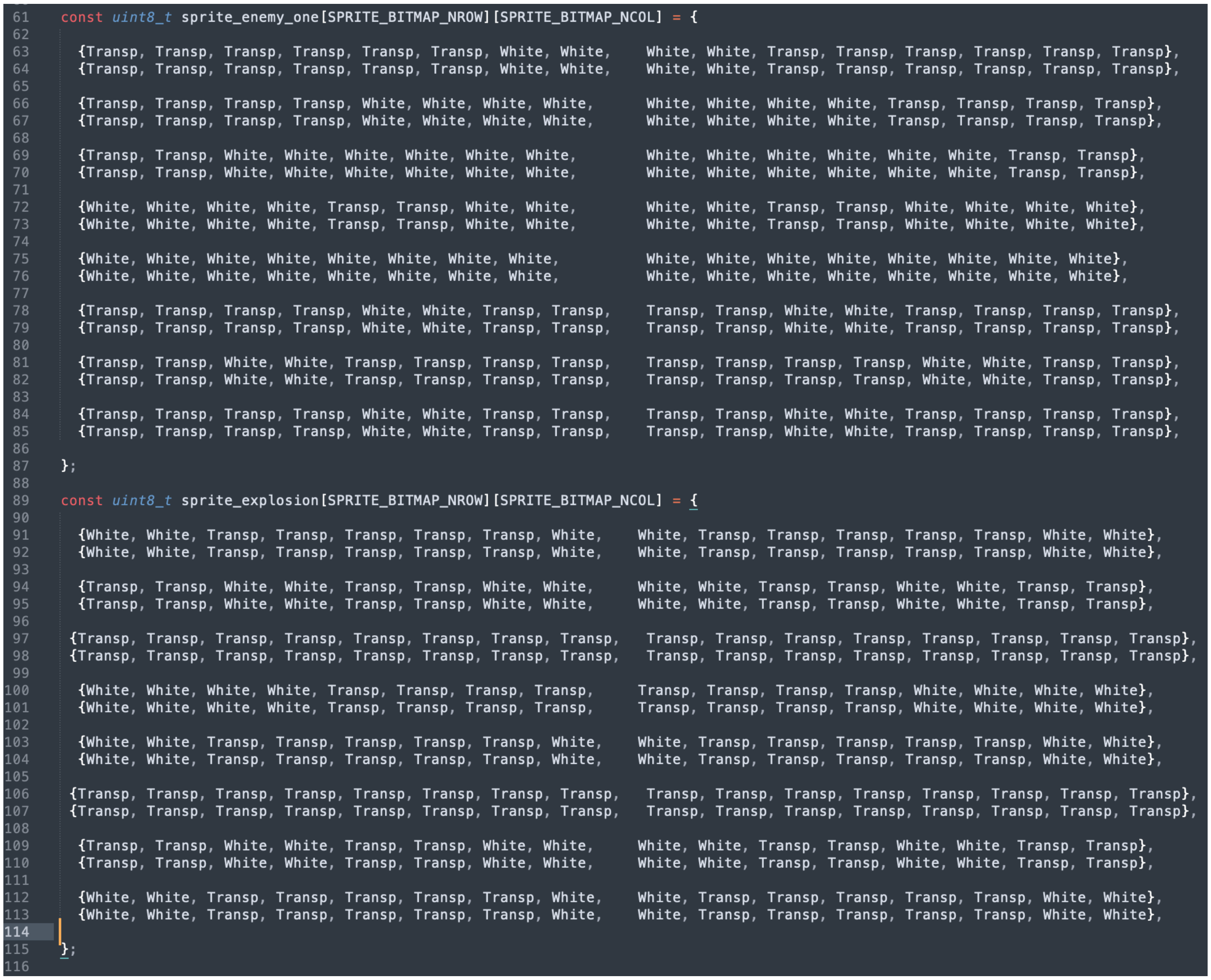


```c
#ifndef _SPRITE_H
#define _SPRITE_H

#include <stdint.h>

typedef struct {
  uint8_t i;
  uint16_t y;
  uint16_t x;
  uint8_t name;
} sprite_attr_t;

#define SPRITE_BITMAP_SIZE 128
#define SPRITE_BITMAP_NROW 16
#define SPRITE_BITMAP_NCOL 16
#define sprite_pixel(x) ((x)&0xf)

void load_sprite_bitmaps();
void set_sprite_bitmap(int spriti, const uint8_t *sprite);
void set_sprite(sprite_attr_t attr);

typedef enum {
  SPRITE_DEFENDER,
  SPRITE_BULLET,
  SPRITE_ENEMY_ONE,
  SPRITE_EXPLOSION,
  SPRITE_TRANSPARENT,
  SPRITE_BOMB,
} sprite_name_t;

#endif
```

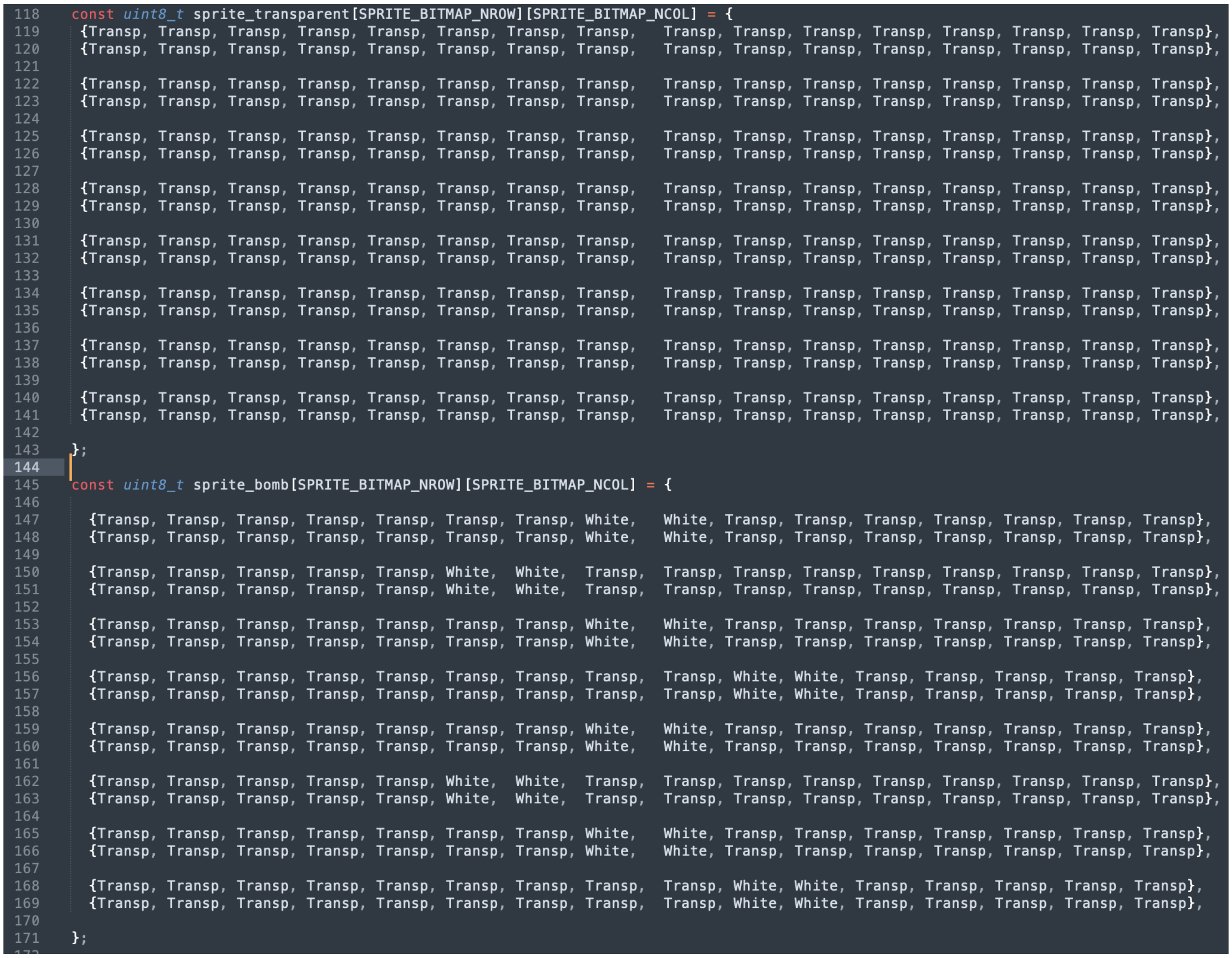
sprite.c



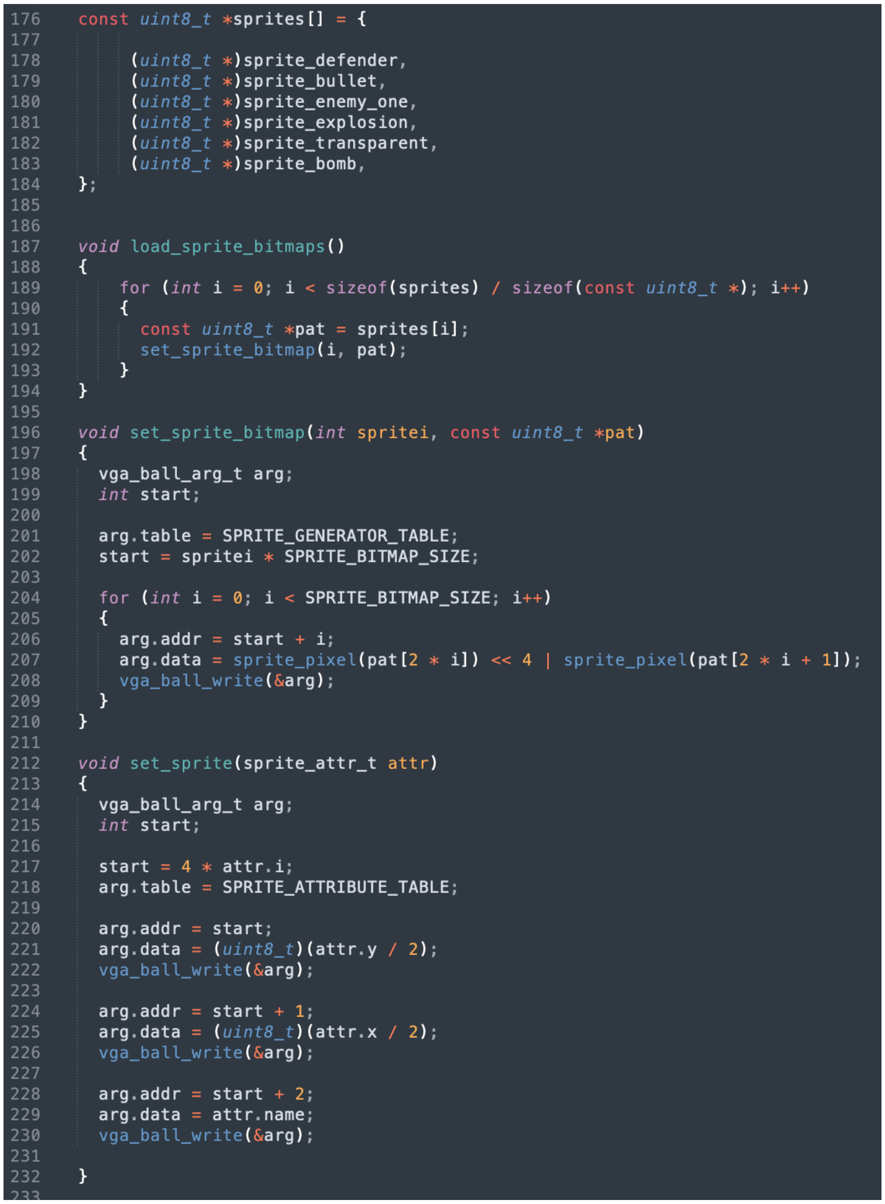
```c
#include "sprite.h"
#include "color.h"
#include "vga_ball_user.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

const uint8_t sprite_defender[SPRITE_BITMAP_NROW][SPRITE_BITMAP_NCOL] = {

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Green, Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Green, Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Green, Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Green,  Green, Green,  Green, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Green,  Green, Green,  Green, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Green,  Green, Green,  Green, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Transp, Transp},
  {Transp, Transp, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Transp, Transp},

  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green},
  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green},
  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green},

  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green},
  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green},
  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green},
  {Green, Green, Green, Green, Green, Green, Green, Green,   Green, Green, Green, Green, Green, Green, Green, Green}

};

const uint8_t sprite_bullet[SPRITE_BITMAP_NROW][SPRITE_BITMAP_NCOL] = {

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Green,   Green, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

};
```



```
const uint8_t sprite_enemy_one[SPRITE_BITMAP_NROW][SPRITE_BITMAP_NCOL] = {

  {Transp, Transp, Transp, Transp, Transp, Transp, White, White,    White, White, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, White, White,    White, White, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, White, White, White, White,      White, White, White, White, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, White, White, White, White,      White, White, White, White, Transp, Transp, Transp, Transp},

  {Transp, Transp, White, White, White, White, White, White,        White, White, White, White, White, White, Transp, Transp},
  {Transp, Transp, White, White, White, White, White, White,        White, White, White, White, White, White, Transp, Transp},

  {White, White, White, White, Transp, Transp, White, White,        White, White, Transp, Transp, White, White, White, White},
  {White, White, White, White, Transp, Transp, White, White,        White, White, Transp, Transp, White, White, White, White},

  {White, White, White, White, White, White, White, White,          White, White, White, White, White, White, White, White},
  {White, White, White, White, White, White, White, White,          White, White, White, White, White, White, White, White},

  {Transp, Transp, Transp, Transp, White, White, Transp, Transp,    Transp, Transp, White, White, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, White, White, Transp, Transp,    Transp, Transp, White, White, Transp, Transp, Transp, Transp},

  {Transp, Transp, White, White, Transp, Transp, Transp, Transp,    Transp, Transp, Transp, Transp, White, White, Transp, Transp},
  {Transp, Transp, White, White, Transp, Transp, Transp, Transp,    Transp, Transp, Transp, Transp, White, White, Transp, Transp},

  {Transp, Transp, Transp, Transp, White, White, Transp, Transp,    Transp, Transp, White, White, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, White, White, Transp, Transp,    Transp, Transp, White, White, Transp, Transp, Transp, Transp},

};

const uint8_t sprite_explosion[SPRITE_BITMAP_NROW][SPRITE_BITMAP_NCOL] = {

  {White, White, Transp, Transp, Transp, Transp, Transp, White,    White, Transp, Transp, Transp, Transp, Transp, White, White},
  {White, White, Transp, Transp, Transp, Transp, Transp, White,    White, Transp, Transp, Transp, Transp, Transp, White, White},

  {Transp, Transp, White, White, Transp, Transp, White, White,     White, White, Transp, Transp, White, White, Transp, Transp},
  {Transp, Transp, White, White, Transp, Transp, White, White,     White, White, Transp, Transp, White, White, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {White, White, White, White, Transp, Transp, Transp, Transp,     Transp, Transp, Transp, Transp, White, White, White, White},
  {White, White, White, White, Transp, Transp, Transp, Transp,     Transp, Transp, Transp, Transp, White, White, White, White},

  {White, White, Transp, Transp, Transp, Transp, Transp, White,    White, Transp, Transp, Transp, Transp, Transp, White, White},
  {White, White, Transp, Transp, Transp, Transp, Transp, White,    White, Transp, Transp, Transp, Transp, Transp, White, White},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, White, White, Transp, Transp, White, White,     White, White, Transp, Transp, White, White, Transp, Transp},
  {Transp, Transp, White, White, Transp, Transp, White, White,     White, White, Transp, Transp, White, White, Transp, Transp},

  {White, White, Transp, Transp, Transp, Transp, Transp, White,    White, Transp, Transp, Transp, Transp, Transp, White, White},
  {White, White, Transp, Transp, Transp, Transp, Transp, White,    White, Transp, Transp, Transp, Transp, Transp, White, White},

};

```

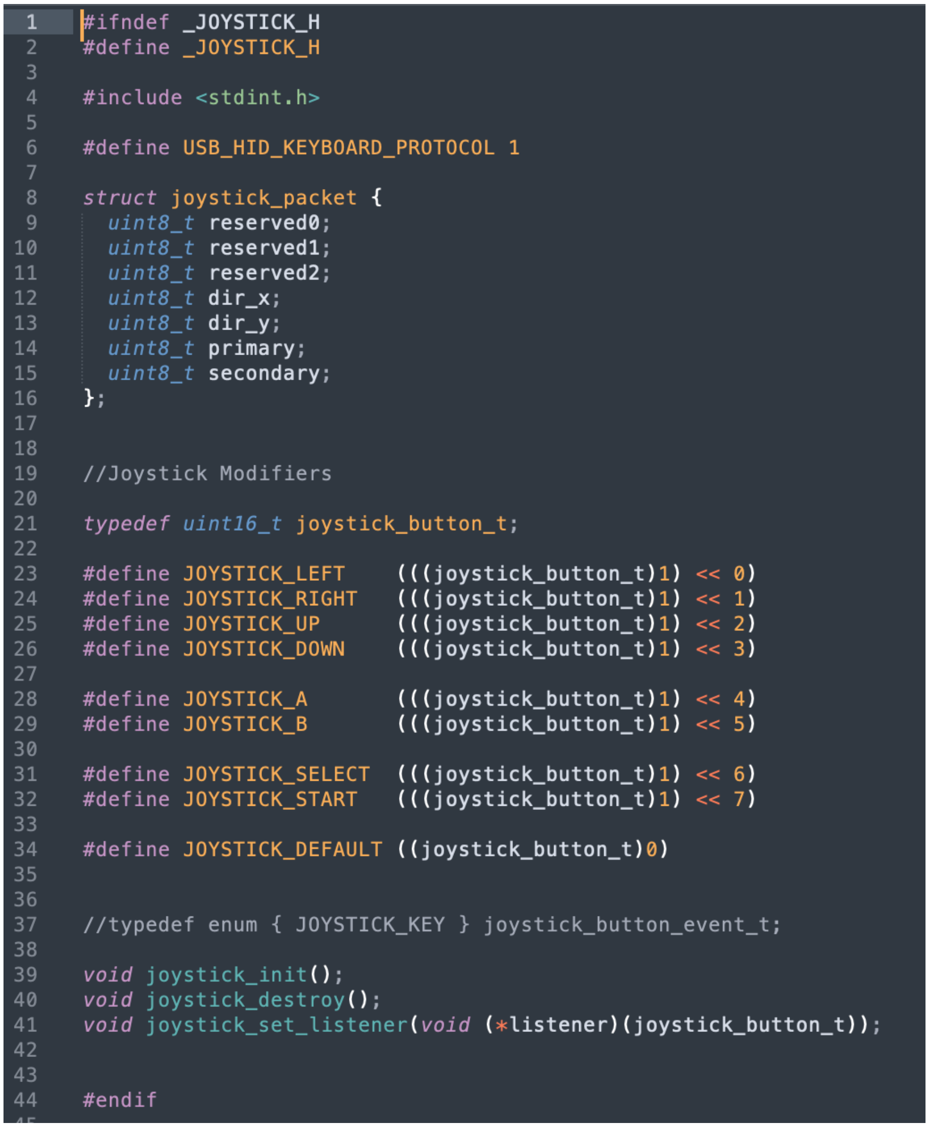


```
const uint8_t sprite_transparent[SPRITE_BITMAP_NROW][SPRITE_BITMAP_NCOL] = {
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
 {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,   Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

};

const uint8_t sprite_bomb[SPRITE_BITMAP_NROW][SPRITE_BITMAP_NCOL] = {

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, White,  White,  Transp,  Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, White,  White,  Transp,  Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Transp, White, White, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Transp, White, White, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, White,  White,  Transp,  Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, White,  White,  Transp,  Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, White,   White, Transp, Transp, Transp, Transp, Transp, Transp, Transp},

  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Transp, White, White, Transp, Transp, Transp, Transp, Transp},
  {Transp, Transp, Transp, Transp, Transp, Transp, Transp, Transp,  Transp, White, White, Transp, Transp, Transp, Transp, Transp},

};
```



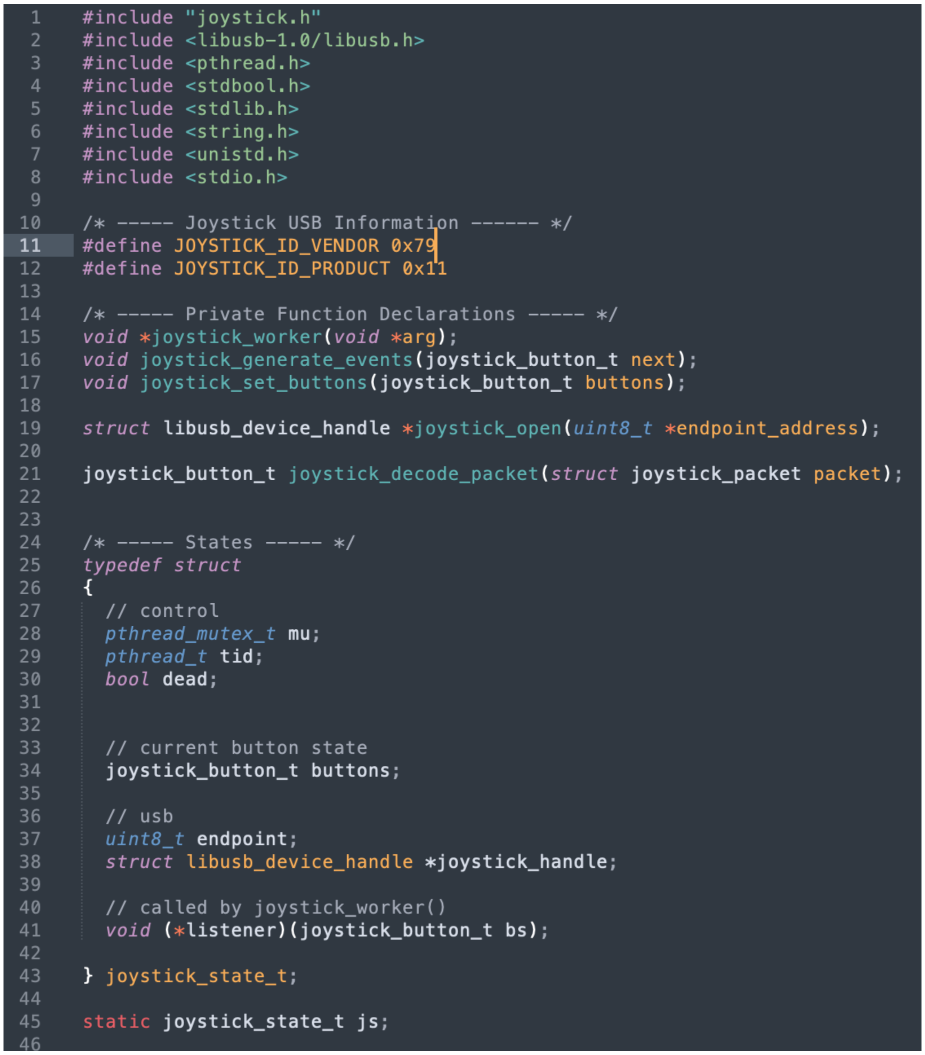
```
const uint8_t *sprites[] = {

     (uint8_t *)sprite_defender,
     (uint8_t *)sprite_bullet,
     (uint8_t *)sprite_enemy_one,
     (uint8_t *)sprite_explosion,
     (uint8_t *)sprite_transparent,
     (uint8_t *)sprite_bomb,
};


void load_sprite_bitmaps()
{
    for (int i = 0; i < sizeof(sprites) / sizeof(const uint8_t *); i++)
    {
      const uint8_t *pat = sprites[i];
      set_sprite_bitmap(i, pat);
    }
}

void set_sprite_bitmap(int spritei, const uint8_t *pat)
{
  vga_ball_arg_t arg;
  int start;

  arg.table = SPRITE_GENERATOR_TABLE;
  start = spritei * SPRITE_BITMAP_SIZE;

  for (int i = 0; i < SPRITE_BITMAP_SIZE; i++)
  {
    arg.addr = start + i;
    arg.data = sprite_pixel(pat[2 * i]) << 4 | sprite_pixel(pat[2 * i + 1]);
    vga_ball_write(&arg);
  }
}

void set_sprite(sprite_attr_t attr)
{
  vga_ball_arg_t arg;
  int start;

  start = 4 * attr.i;
  arg.table = SPRITE_ATTRIBUTE_TABLE;

  arg.addr = start;
  arg.data = (uint8_t)(attr.y / 2);
  vga_ball_write(&arg);

  arg.addr = start + 1;
  arg.data = (uint8_t)(attr.x / 2);
  vga_ball_write(&arg);

  arg.addr = start + 2;
  arg.data = attr.name;
  vga_ball_write(&arg);

}
```

joystick.h

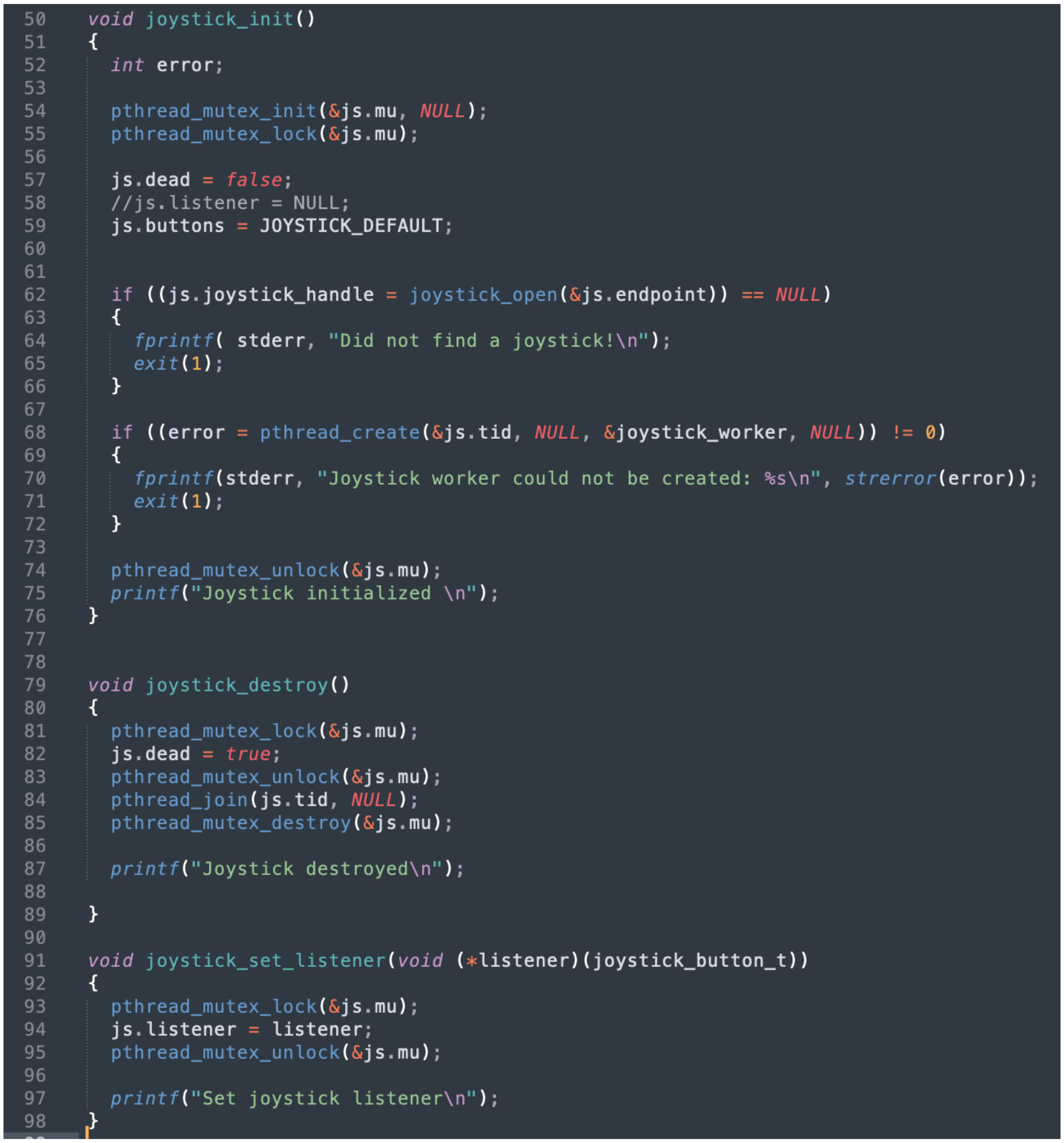


```c
#ifndef _JOYSTICK_H
#define _JOYSTICK_H

#include <stdint.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

struct joystick_packet {
  uint8_t reserved0;
  uint8_t reserved1;
  uint8_t reserved2;
  uint8_t dir_x;
  uint8_t dir_y;
  uint8_t primary;
  uint8_t secondary;
};


//Joystick Modifiers

typedef uint16_t joystick_button_t;

#define JOYSTICK_LEFT    (((joystick_button_t)1) << 0)
#define JOYSTICK_RIGHT   (((joystick_button_t)1) << 1)
#define JOYSTICK_UP      (((joystick_button_t)1) << 2)
#define JOYSTICK_DOWN    (((joystick_button_t)1) << 3)

#define JOYSTICK_A       (((joystick_button_t)1) << 4)
#define JOYSTICK_B       (((joystick_button_t)1) << 5)

#define JOYSTICK_SELECT  (((joystick_button_t)1) << 6)
#define JOYSTICK_START   (((joystick_button_t)1) << 7)

#define JOYSTICK_DEFAULT ((joystick_button_t)0)


//typedef enum { JOYSTICK_KEY } joystick_button_event_t;

void joystick_init();
void joystick_destroy();
void joystick_set_listener(void (*listener)(joystick_button_t));


#endif
```

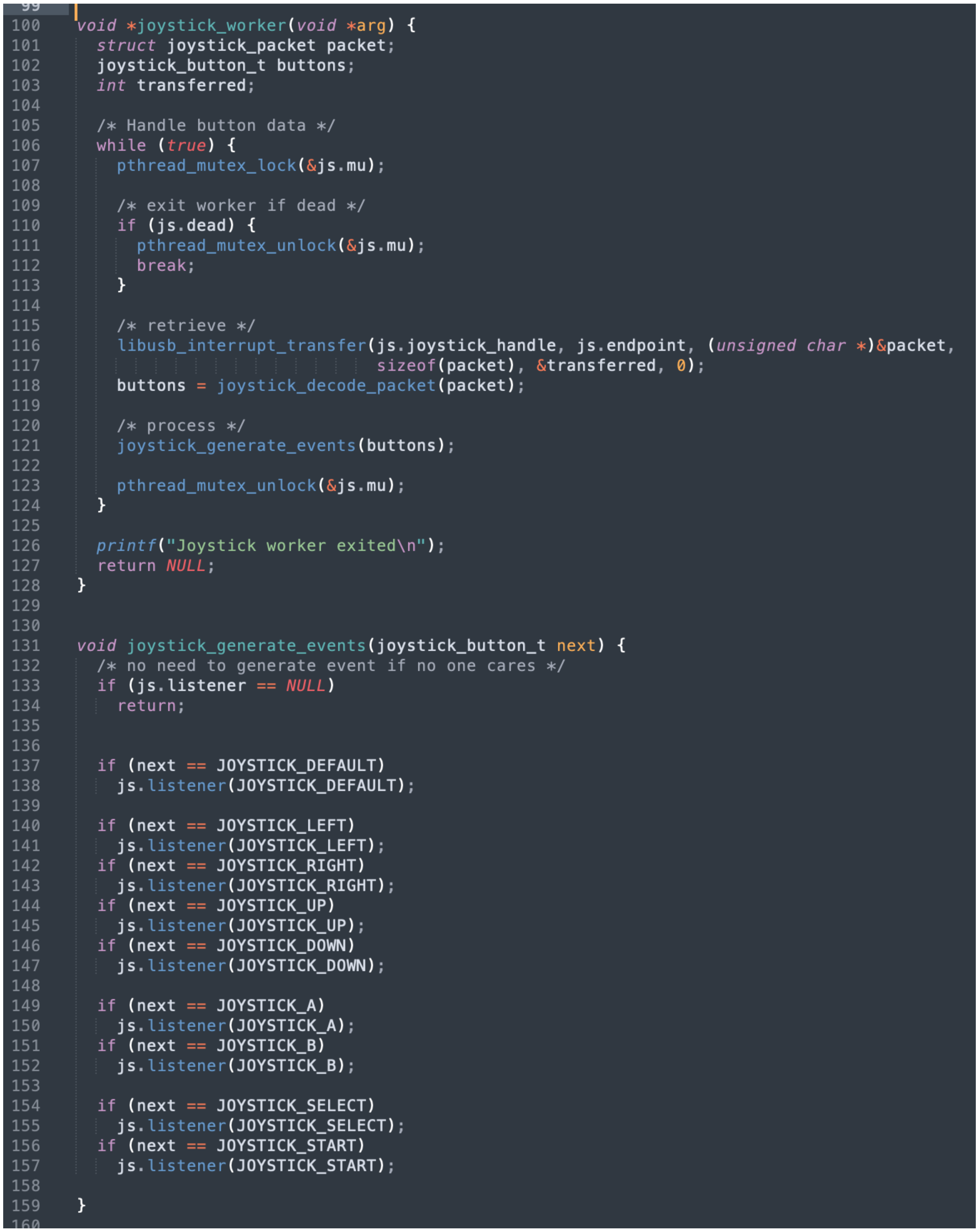
joystick.c



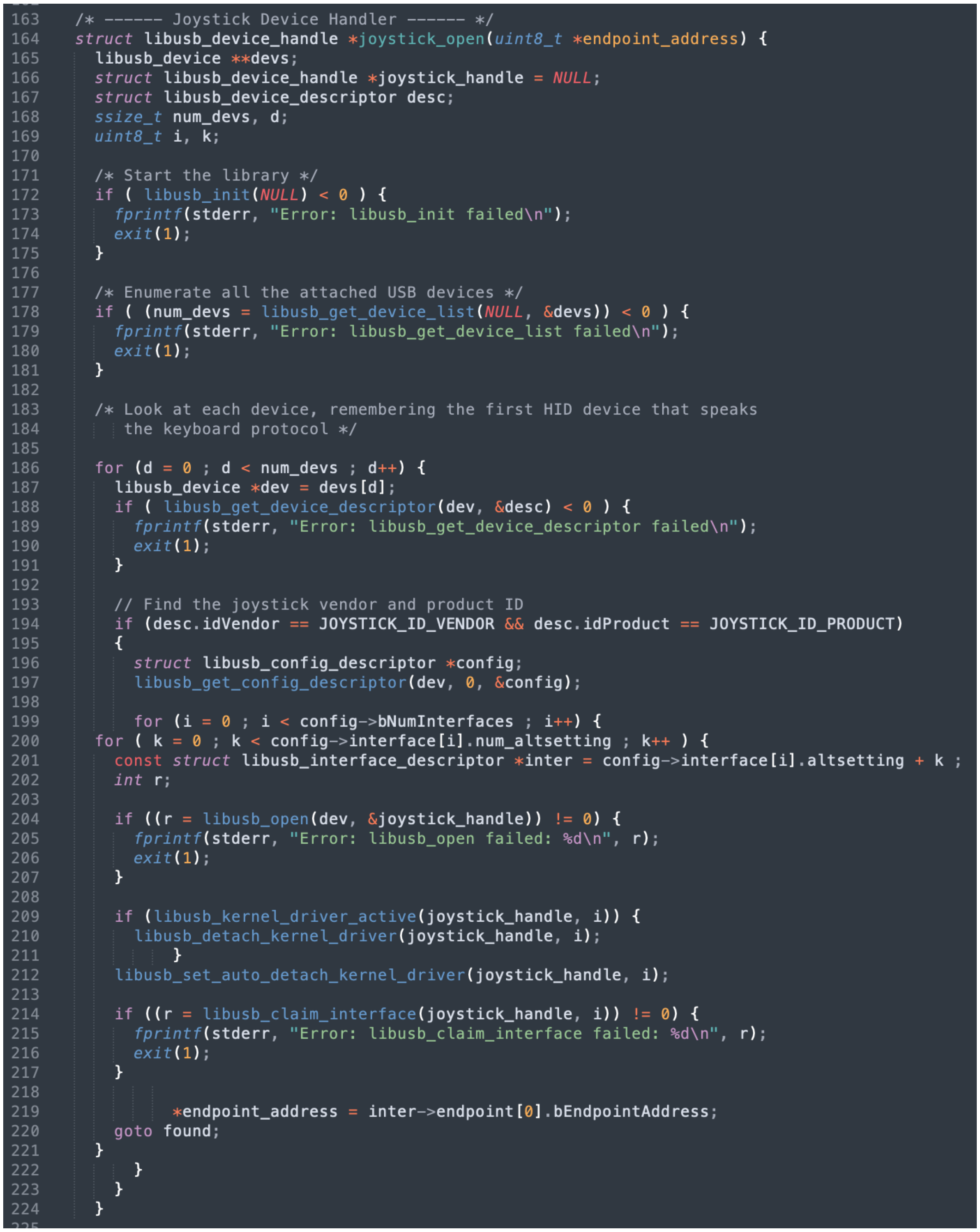
```c
#include "joystick.h"
#include <libusb-1.0/libusb.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

/* ----- Joystick USB Information ------ */
#define JOYSTICK_ID_VENDOR 0x79
#define JOYSTICK_ID_PRODUCT 0x11

/* ----- Private Function Declarations ----- */
void *joystick_worker(void *arg);
void joystick_generate_events(joystick_button_t next);
void joystick_set_buttons(joystick_button_t buttons);

struct libusb_device_handle *joystick_open(uint8_t *endpoint_address);

joystick_button_t joystick_decode_packet(struct joystick_packet packet);


/* ----- States ----- */
typedef struct
{
  // control
  pthread_mutex_t mu;
  pthread_t tid;
  bool dead;


  // current button state
  joystick_button_t buttons;

  // usb
  uint8_t endpoint;
  struct libusb_device_handle *joystick_handle;

  // called by joystick_worker()
  void (*listener)(joystick_button_t bs);

} joystick_state_t;

static joystick_state_t js;
```



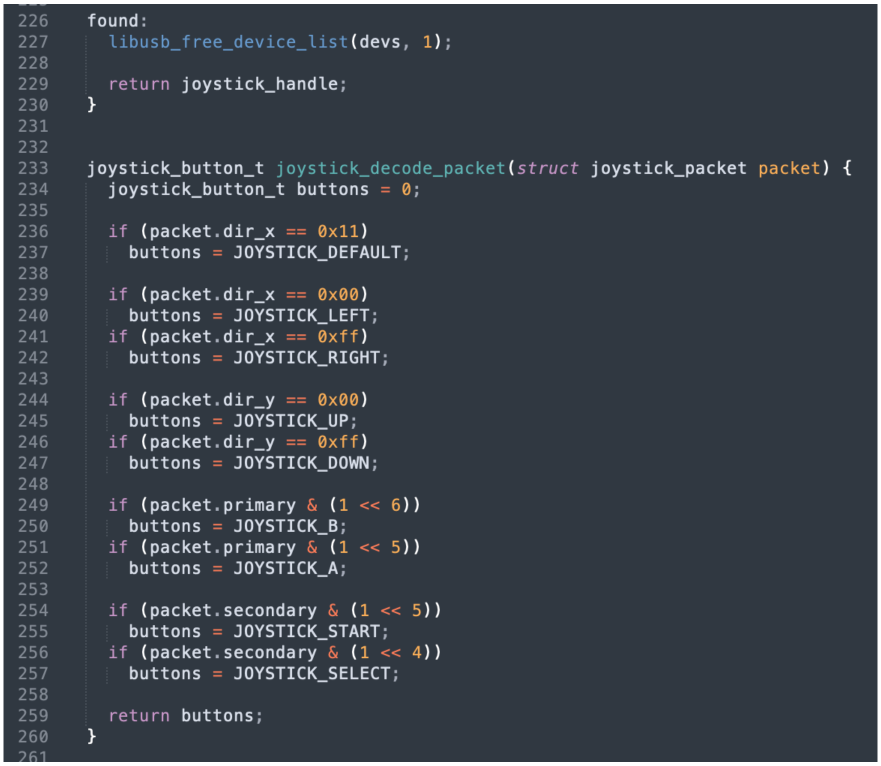
```
void joystick_init()
{
  int error;

  pthread_mutex_init(&js.mu, NULL);
  pthread_mutex_lock(&js.mu);

  js.dead = false;
  //js.listener = NULL;
  js.buttons = JOYSTICK_DEFAULT;


  if ((js.joystick_handle = joystick_open(&js.endpoint)) == NULL)
  {
    fprintf( stderr, "Did not find a joystick!\n");
    exit(1);
  }

  if ((error = pthread_create(&js.tid, NULL, &joystick_worker, NULL)) != 0)
  {
    fprintf(stderr, "Joystick worker could not be created: %s\n", strerror(error));
    exit(1);
  }

  pthread_mutex_unlock(&js.mu);
  printf("Joystick initialized \n");
}


void joystick_destroy()
{
  pthread_mutex_lock(&js.mu);
  js.dead = true;
  pthread_mutex_unlock(&js.mu);
  pthread_join(js.tid, NULL);
  pthread_mutex_destroy(&js.mu);

  printf("Joystick destroyed\n");

}

void joystick_set_listener(void (*listener)(joystick_button_t))
{
  pthread_mutex_lock(&js.mu);
  js.listener = listener;
  pthread_mutex_unlock(&js.mu);

  printf("Set joystick listener\n");
}
```



```c
void *joystick_worker(void *arg) {
  struct joystick_packet packet;
  joystick_button_t buttons;
  int transferred;

  /* Handle button data */
  while (true) {
    pthread_mutex_lock(&js.mu);

    /* exit worker if dead */
    if (js.dead) {
      pthread_mutex_unlock(&js.mu);
      break;
    }

    /* retrieve */
    libusb_interrupt_transfer(js.joystick_handle, js.endpoint, (unsigned char *)&packet,
                              sizeof(packet), &transferred, 0);
    buttons = joystick_decode_packet(packet);

    /* process */
    joystick_generate_events(buttons);

    pthread_mutex_unlock(&js.mu);
  }

  printf("Joystick worker exited\n");
  return NULL;
}


void joystick_generate_events(joystick_button_t next) {
  /* no need to generate event if no one cares */
  if (js.listener == NULL)
    return;


  if (next == JOYSTICK_DEFAULT)
    js.listener(JOYSTICK_DEFAULT);

  if (next == JOYSTICK_LEFT)
    js.listener(JOYSTICK_LEFT);
  if (next == JOYSTICK_RIGHT)
    js.listener(JOYSTICK_RIGHT);
  if (next == JOYSTICK_UP)
    js.listener(JOYSTICK_UP);
  if (next == JOYSTICK_DOWN)
    js.listener(JOYSTICK_DOWN);

  if (next == JOYSTICK_A)
    js.listener(JOYSTICK_A);
  if (next == JOYSTICK_B)
    js.listener(JOYSTICK_B);

  if (next == JOYSTICK_SELECT)
    js.listener(JOYSTICK_SELECT);
  if (next == JOYSTICK_START)
    js.listener(JOYSTICK_START);

}
```

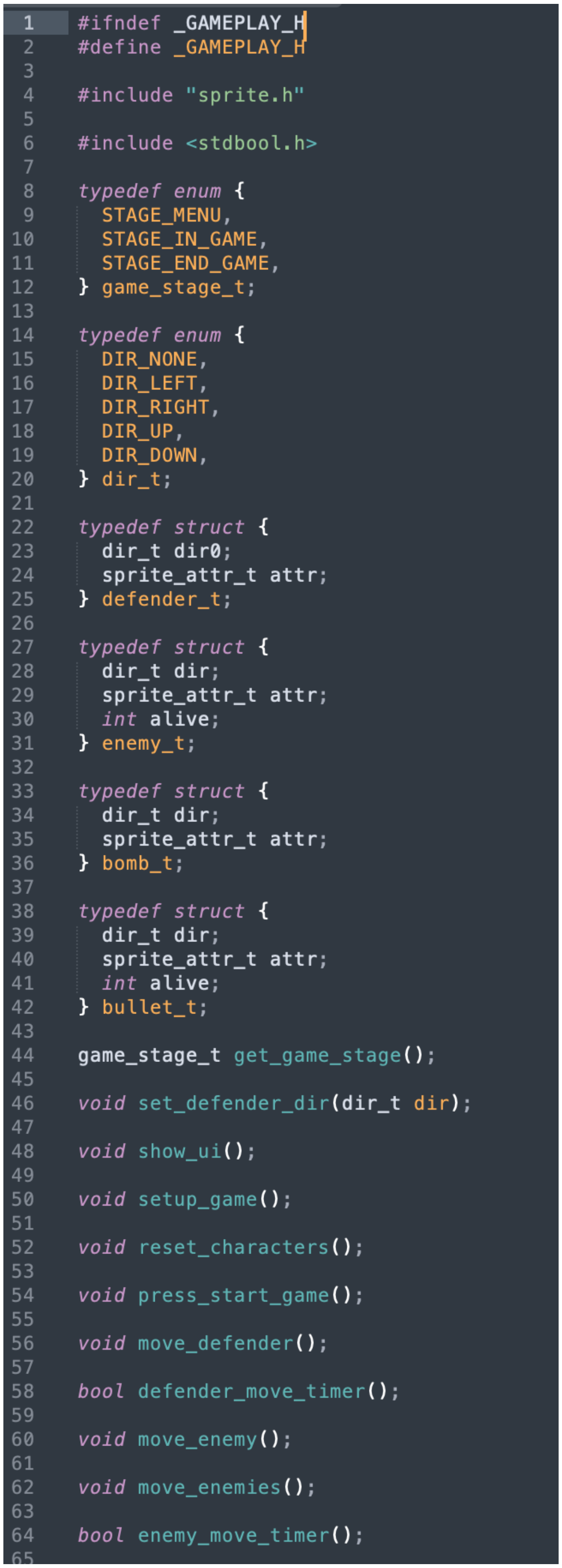


```c
/* ------ Joystick Device Handler ------ */
struct libusb_device_handle *joystick_open(uint8_t *endpoint_address) {
  libusb_device **devs;
  struct libusb_device_handle *joystick_handle = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  /* Start the library */
  if ( libusb_init(NULL) < 0 ) {
    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
  }

  /* Enumerate all the attached USB devices */
  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
  }

  /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }

    // Find the joystick vendor and product ID
    if (desc.idVendor == JOYSTICK_ID_VENDOR && desc.idProduct == JOYSTICK_ID_PRODUCT)
    {
      struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);

      for (i = 0 ; i < config->bNumInterfaces ; i++) {
  for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
    const struct libusb_interface_descriptor *inter = config->interface[i].altsetting + k ;
    int r;

    if ((r = libusb_open(dev, &joystick_handle)) != 0) {
      fprintf(stderr, "Error: libusb_open failed: %d\n", r);
      exit(1);
    }

    if (libusb_kernel_driver_active(joystick_handle, i)) {
      libusb_detach_kernel_driver(joystick_handle, i);
          }
    libusb_set_auto_detach_kernel_driver(joystick_handle, i);

    if ((r = libusb_claim_interface(joystick_handle, i)) != 0) {
      fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
      exit(1);
    }

          *endpoint_address = inter->endpoint[0].bEndpointAddress;
    goto found;
  }
      }
    }
  }
```

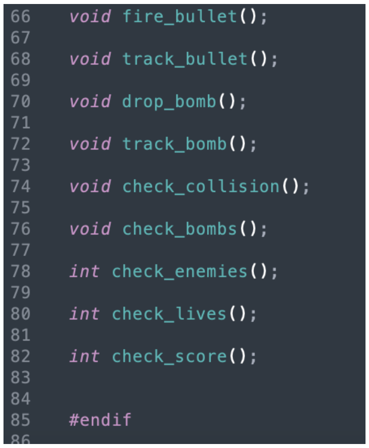


```c
found:
  libusb_free_device_list(devs, 1);

  return joystick_handle;
}


joystick_button_t joystick_decode_packet(struct joystick_packet packet) {
  joystick_button_t buttons = 0;

  if (packet.dir_x == 0x11)
    buttons = JOYSTICK_DEFAULT;

  if (packet.dir_x == 0x00)
    buttons = JOYSTICK_LEFT;
  if (packet.dir_x == 0xff)
    buttons = JOYSTICK_RIGHT;

  if (packet.dir_y == 0x00)
    buttons = JOYSTICK_UP;
  if (packet.dir_y == 0xff)
    buttons = JOYSTICK_DOWN;

  if (packet.primary & (1 << 6))
    buttons = JOYSTICK_B;
  if (packet.primary & (1 << 5))
    buttons = JOYSTICK_A;

  if (packet.secondary & (1 << 5))
    buttons = JOYSTICK_START;
  if (packet.secondary & (1 << 4))
    buttons = JOYSTICK_SELECT;

  return buttons;
}
```

gameplay.h

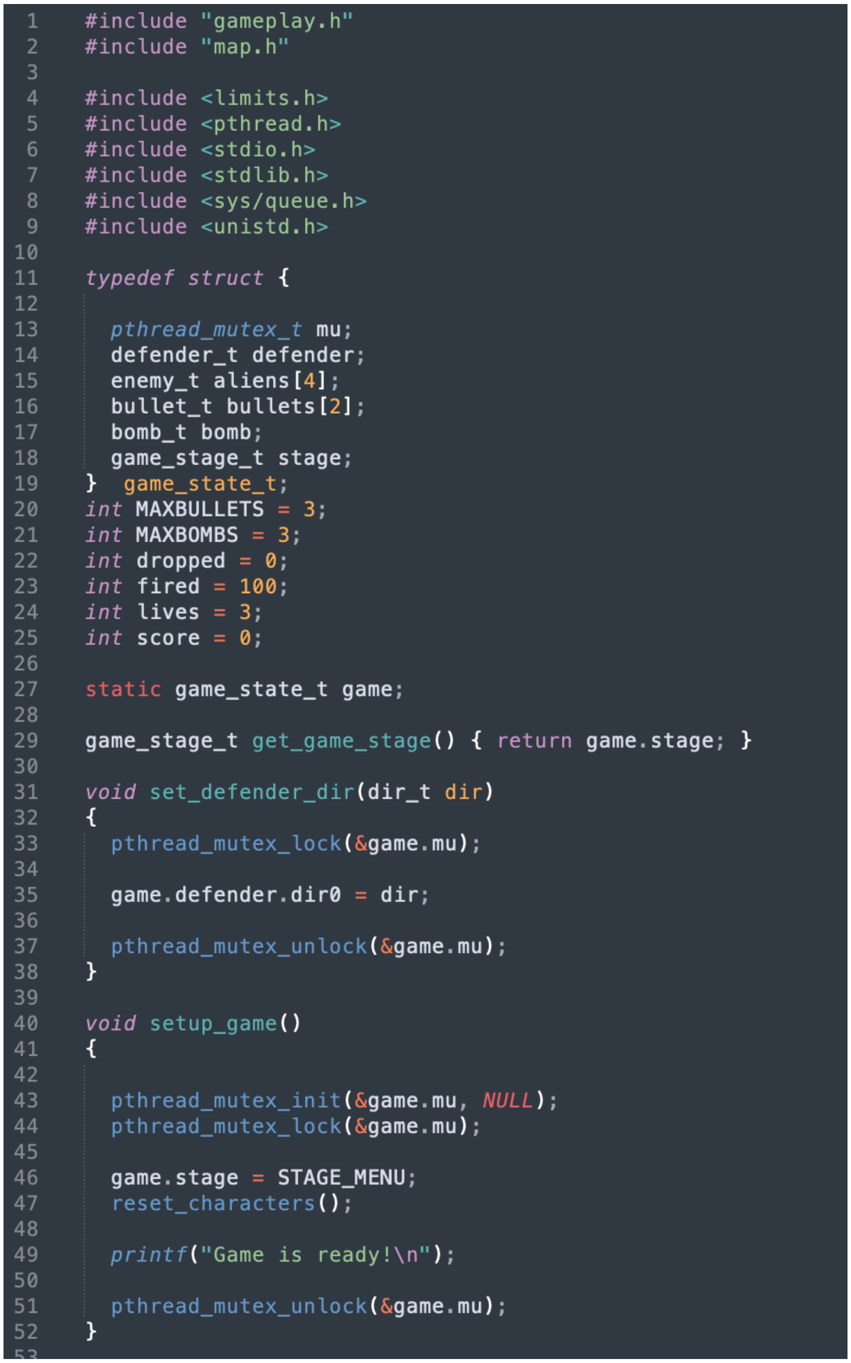


```c
#ifndef _GAMEPLAY_H
#define _GAMEPLAY_H

#include "sprite.h"

#include <stdbool.h>

typedef enum {
  STAGE_MENU,
  STAGE_IN_GAME,
  STAGE_END_GAME,
} game_stage_t;

typedef enum {
  DIR_NONE,
  DIR_LEFT,
  DIR_RIGHT,
  DIR_UP,
  DIR_DOWN,
} dir_t;

typedef struct {
  dir_t dir0;
  sprite_attr_t attr;
} defender_t;

typedef struct {
  dir_t dir;
  sprite_attr_t attr;
  int alive;
} enemy_t;

typedef struct {
  dir_t dir;
  sprite_attr_t attr;
} bomb_t;

typedef struct {
  dir_t dir;
  sprite_attr_t attr;
  int alive;
} bullet_t;

game_stage_t get_game_stage();

void set_defender_dir(dir_t dir);

void show_ui();

void setup_game();

void reset_characters();

void press_start_game();

void move_defender();

bool defender_move_timer();

void move_enemy();

void move_enemies();

bool enemy_move_timer();

```

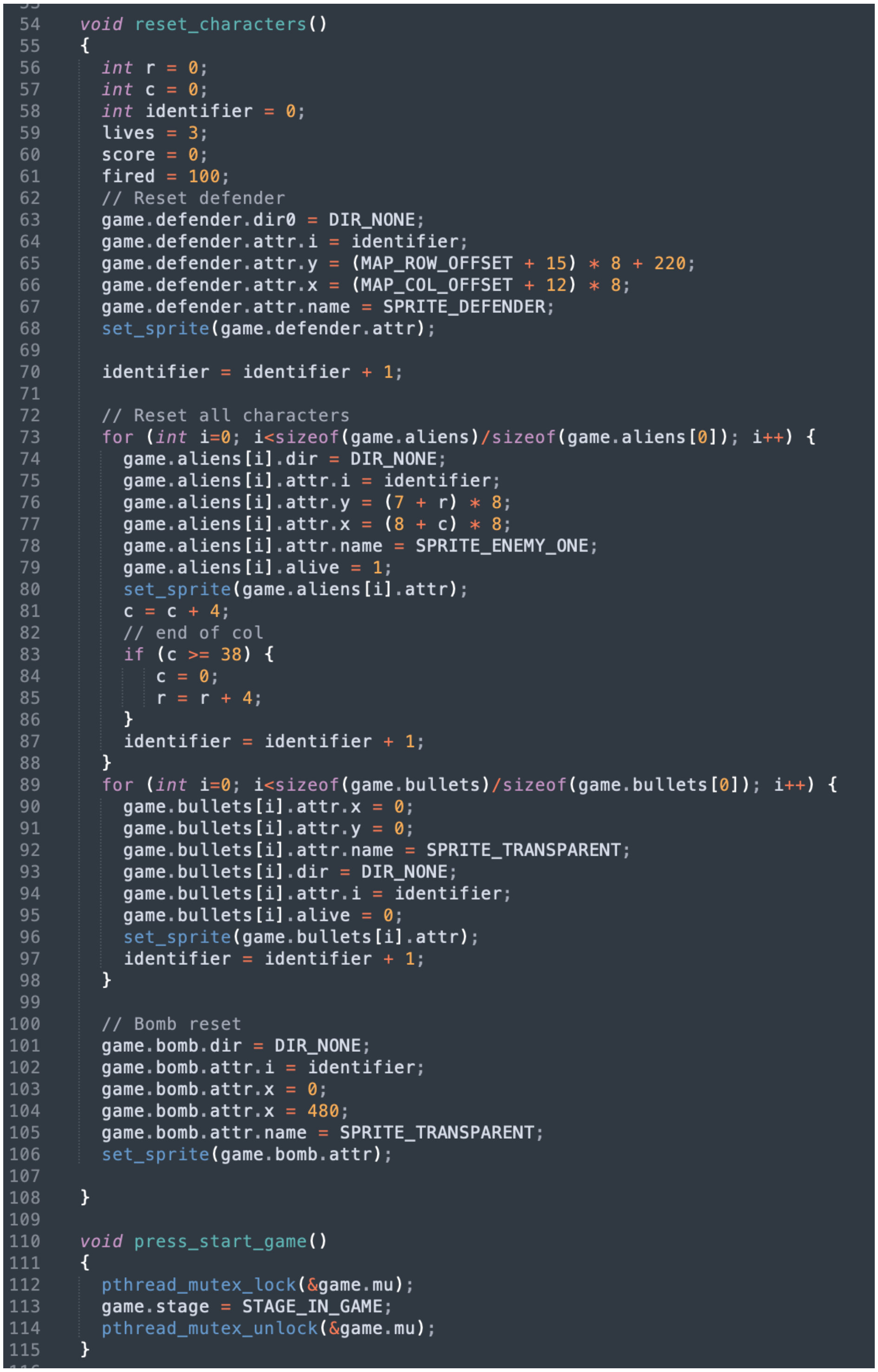


```
void fire_bullet();

void track_bullet();

void drop_bomb();

void track_bomb();

void check_collision();

void check_bombs();

int check_enemies();

int check_lives();

int check_score();


#endif
```

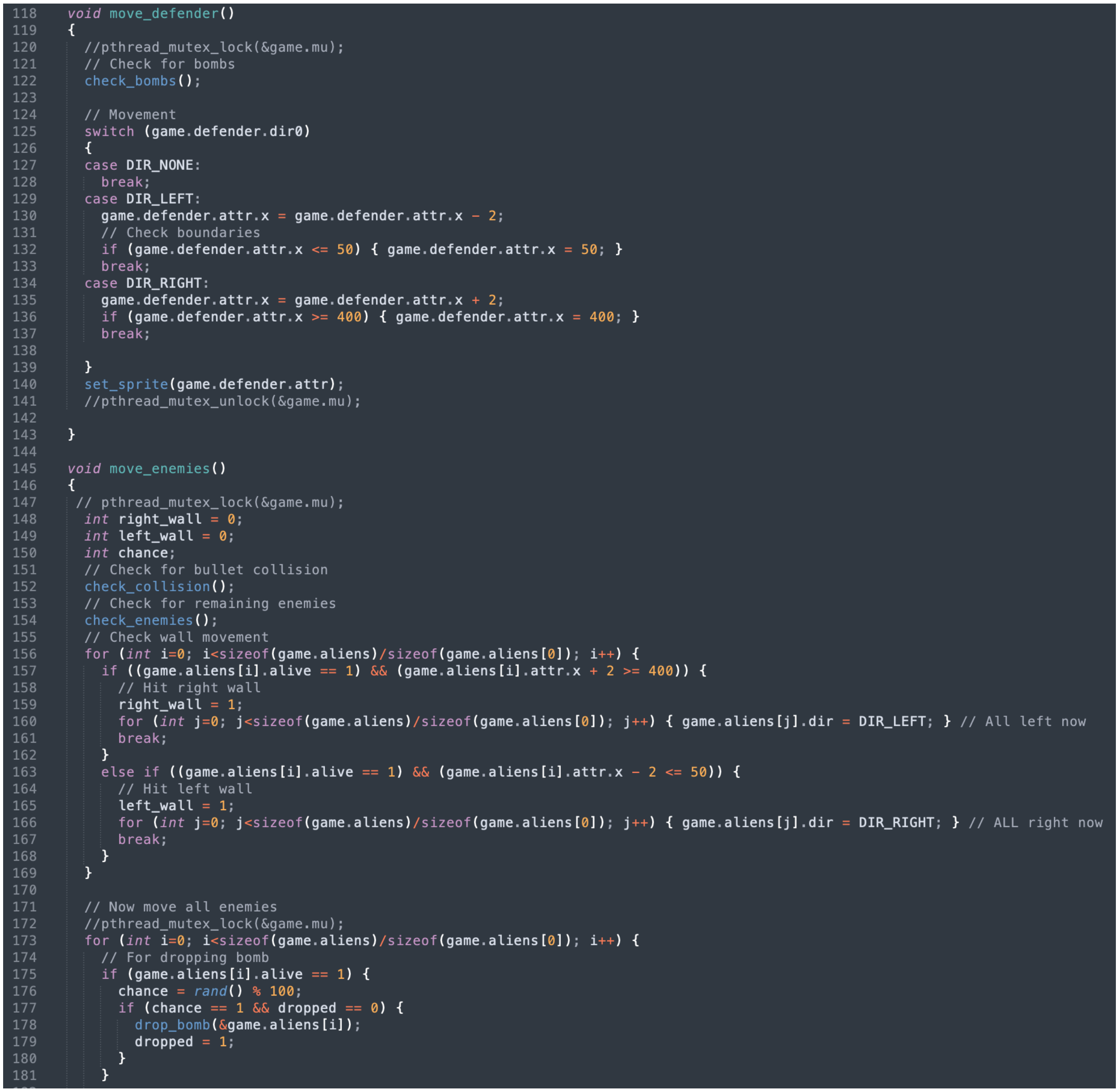
gameplay.c



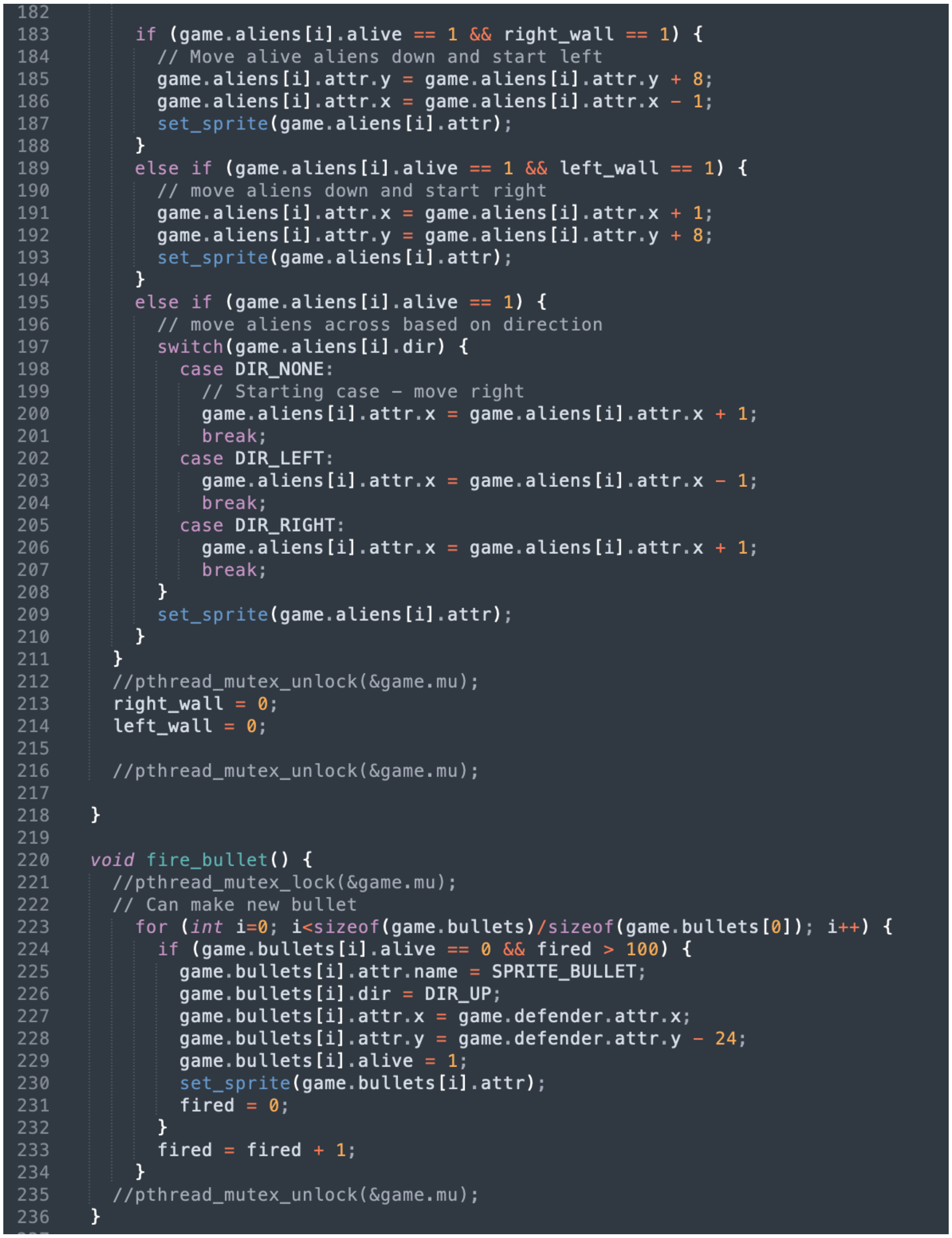
```c
#include "gameplay.h"
#include "map.h"

#include <limits.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>
#include <unistd.h>

typedef struct {

  pthread_mutex_t mu;
  defender_t defender;
  enemy_t aliens[4];
  bullet_t bullets[2];
  bomb_t bomb;
  game_stage_t stage;
}  game_state_t;
int MAXBULLETS = 3;
int MAXBOMBS = 3;
int dropped = 0;
int fired = 100;
int lives = 3;
int score = 0;

static game_state_t game;

game_stage_t get_game_stage() { return game.stage; }

void set_defender_dir(dir_t dir)
{
  pthread_mutex_lock(&game.mu);

  game.defender.dir0 = dir;

  pthread_mutex_unlock(&game.mu);
}

void setup_game()
{

  pthread_mutex_init(&game.mu, NULL);
  pthread_mutex_lock(&game.mu);

  game.stage = STAGE_MENU;
  reset_characters();

  printf("Game is ready!\n");

  pthread_mutex_unlock(&game.mu);
}
```



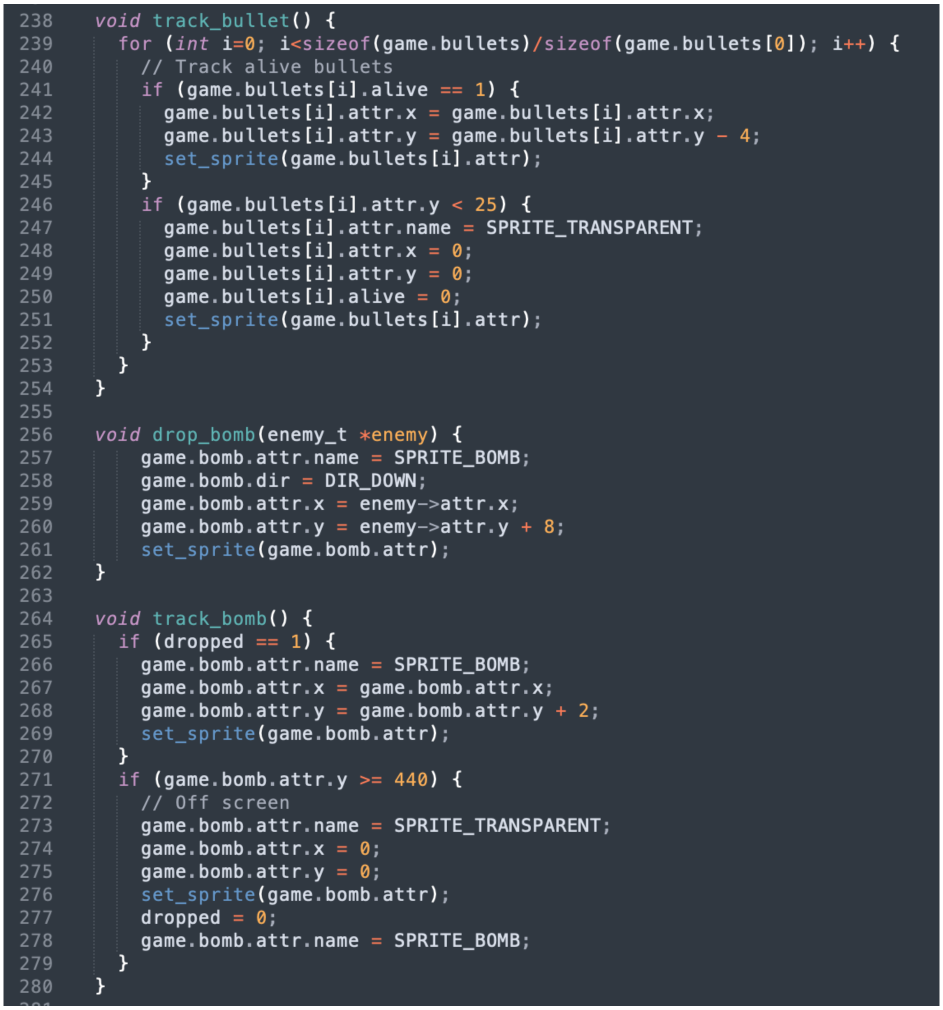
```c
void reset_characters()
{
  int r = 0;
  int c = 0;
  int identifier = 0;
  lives = 3;
  score = 0;
  fired = 100;
  // Reset defender
  game.defender.dir0 = DIR_NONE;
  game.defender.attr.i = identifier;
  game.defender.attr.y = (MAP_ROW_OFFSET + 15) * 8 + 220;
  game.defender.attr.x = (MAP_COL_OFFSET + 12) * 8;
  game.defender.attr.name = SPRITE_DEFENDER;
  set_sprite(game.defender.attr);

  identifier = identifier + 1;

  // Reset all characters
  for (int i=0; i<sizeof(game.aliens)/sizeof(game.aliens[0]); i++) {
    game.aliens[i].dir = DIR_NONE;
    game.aliens[i].attr.i = identifier;
    game.aliens[i].attr.y = (7 + r) * 8;
    game.aliens[i].attr.x = (8 + c) * 8;
    game.aliens[i].attr.name = SPRITE_ENEMY_ONE;
    game.aliens[i].alive = 1;
    set_sprite(game.aliens[i].attr);
    c = c + 4;
    // end of col
    if (c >= 38) {
      c = 0;
      r = r + 4;
    }
    identifier = identifier + 1;
  }
  for (int i=0; i<sizeof(game.bullets)/sizeof(game.bullets[0]); i++) {
    game.bullets[i].attr.x = 0;
    game.bullets[i].attr.y = 0;
    game.bullets[i].attr.name = SPRITE_TRANSPARENT;
    game.bullets[i].dir = DIR_NONE;
    game.bullets[i].attr.i = identifier;
    game.bullets[i].alive = 0;
    set_sprite(game.bullets[i].attr);
    identifier = identifier + 1;
  }

  // Bomb reset
  game.bomb.dir = DIR_NONE;
  game.bomb.attr.i = identifier;
  game.bomb.attr.x = 0;
  game.bomb.attr.x = 480;
  game.bomb.attr.name = SPRITE_TRANSPARENT;
  set_sprite(game.bomb.attr);

}

void press_start_game()
{
  pthread_mutex_lock(&game.mu);
  game.stage = STAGE_IN_GAME;
  pthread_mutex_unlock(&game.mu);
}
```



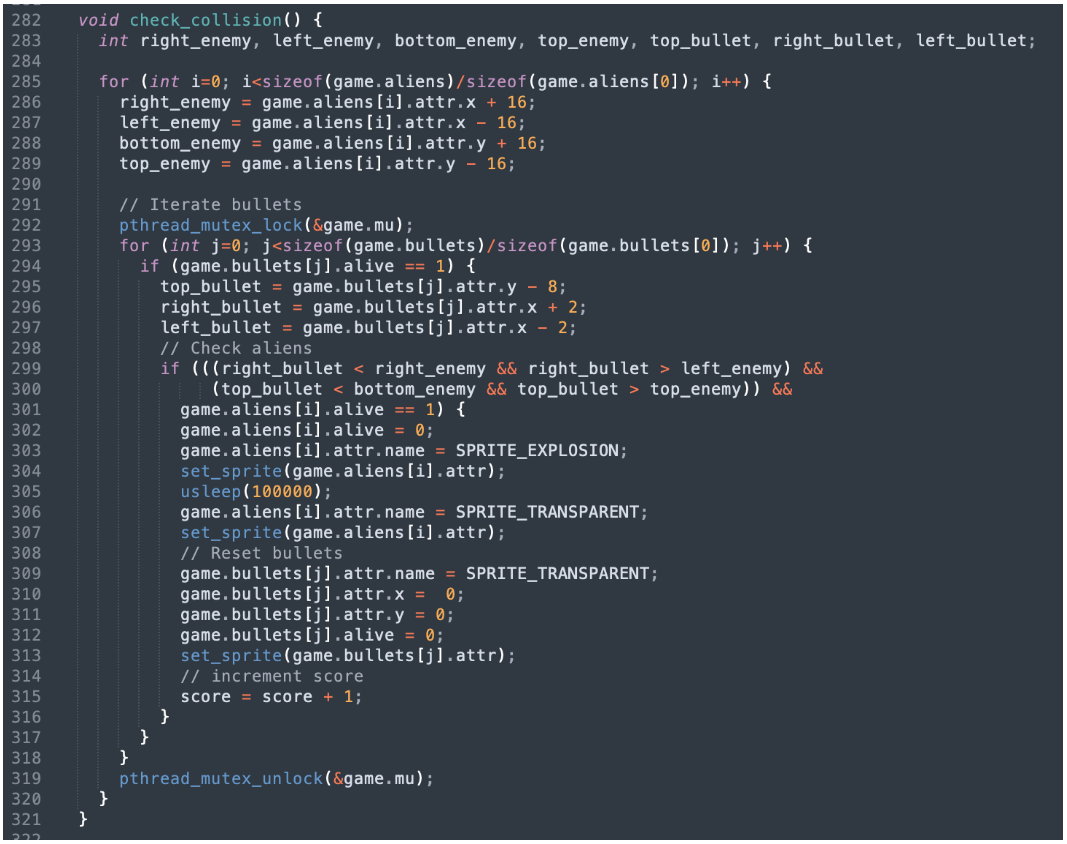
```c
void move_defender()
{
  //pthread_mutex_lock(&game.mu);
  // Check for bombs
  check_bombs();

  // Movement
  switch (game.defender.dir0)
  {
  case DIR_NONE:
    break;
  case DIR_LEFT:
    game.defender.attr.x = game.defender.attr.x - 2;
    // Check boundaries
    if (game.defender.attr.x <= 50) { game.defender.attr.x = 50; }
    break;
  case DIR_RIGHT:
    game.defender.attr.x = game.defender.attr.x + 2;
    if (game.defender.attr.x >= 400) { game.defender.attr.x = 400; }
    break;

  }
  set_sprite(game.defender.attr);
  //pthread_mutex_unlock(&game.mu);

}

void move_enemies()
{
 // pthread_mutex_lock(&game.mu);
  int right_wall = 0;
  int left_wall = 0;
  int chance;
  // Check for bullet collision
  check_collision();
  // Check for remaining enemies
  check_enemies();
  // Check wall movement
  for (int i=0; i<sizeof(game.aliens)/sizeof(game.aliens[0]); i++) {
    if ((game.aliens[i].alive == 1) && (game.aliens[i].attr.x + 2 >= 400)) {
      // Hit right wall
      right_wall = 1;
      for (int j=0; j<sizeof(game.aliens)/sizeof(game.aliens[0]); j++) { game.aliens[j].dir = DIR_LEFT; } // All left now
      break;
    }
    else if ((game.aliens[i].alive == 1) && (game.aliens[i].attr.x - 2 <= 50)) {
      // Hit left wall
      left_wall = 1;
      for (int j=0; j<sizeof(game.aliens)/sizeof(game.aliens[0]); j++) { game.aliens[j].dir = DIR_RIGHT; } // ALL right now
      break;
    }
  }

  // Now move all enemies
  //pthread_mutex_lock(&game.mu);
  for (int i=0; i<sizeof(game.aliens)/sizeof(game.aliens[0]); i++) {
    // For dropping bomb
    if (game.aliens[i].alive == 1) {
      chance = rand() % 100;
      if (chance == 1 && dropped == 0) {
        drop_bomb(&game.aliens[i]);
        dropped = 1;
      }
    }
```



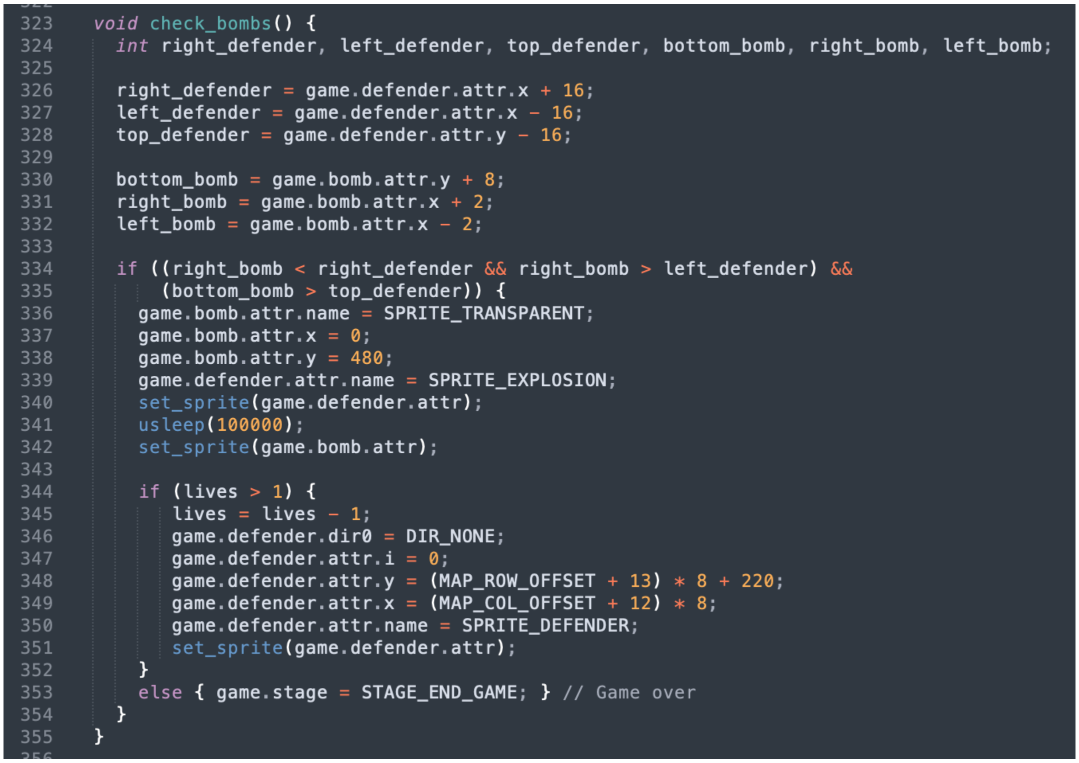
```
        if (game.aliens[i].alive == 1 && right_wall == 1) {
            // Move alive aliens down and start left
            game.aliens[i].attr.y = game.aliens[i].attr.y + 8;
            game.aliens[i].attr.x = game.aliens[i].attr.x - 1;
            set_sprite(game.aliens[i].attr);
        }
        else if (game.aliens[i].alive == 1 && left_wall == 1) {
            // move aliens down and start right
            game.aliens[i].attr.x = game.aliens[i].attr.x + 1;
            game.aliens[i].attr.y = game.aliens[i].attr.y + 8;
            set_sprite(game.aliens[i].attr);
        }
        else if (game.aliens[i].alive == 1) {
            // move aliens across based on direction
            switch(game.aliens[i].dir) {
                case DIR_NONE:
                    // Starting case - move right
                    game.aliens[i].attr.x = game.aliens[i].attr.x + 1;
                    break;
                case DIR_LEFT:
                    game.aliens[i].attr.x = game.aliens[i].attr.x - 1;
                    break;
                case DIR_RIGHT:
                    game.aliens[i].attr.x = game.aliens[i].attr.x + 1;
                    break;
            }
            set_sprite(game.aliens[i].attr);
        }
    }
    //pthread_mutex_unlock(&game.mu);
    right_wall = 0;
    left_wall = 0;

    //pthread_mutex_unlock(&game.mu);

}

void fire_bullet() {
    //pthread_mutex_lock(&game.mu);
    // Can make new bullet
        for (int i=0; i<sizeof(game.bullets)/sizeof(game.bullets[0]); i++) {
            if (game.bullets[i].alive == 0 && fired > 100) {
                game.bullets[i].attr.name = SPRITE_BULLET;
                game.bullets[i].dir = DIR_UP;
                game.bullets[i].attr.x = game.defender.attr.x;
                game.bullets[i].attr.y = game.defender.attr.y - 24;
                game.bullets[i].alive = 1;
                set_sprite(game.bullets[i].attr);
                fired = 0;
            }
            fired = fired + 1;
        }
    //pthread_mutex_unlock(&game.mu);
}
```



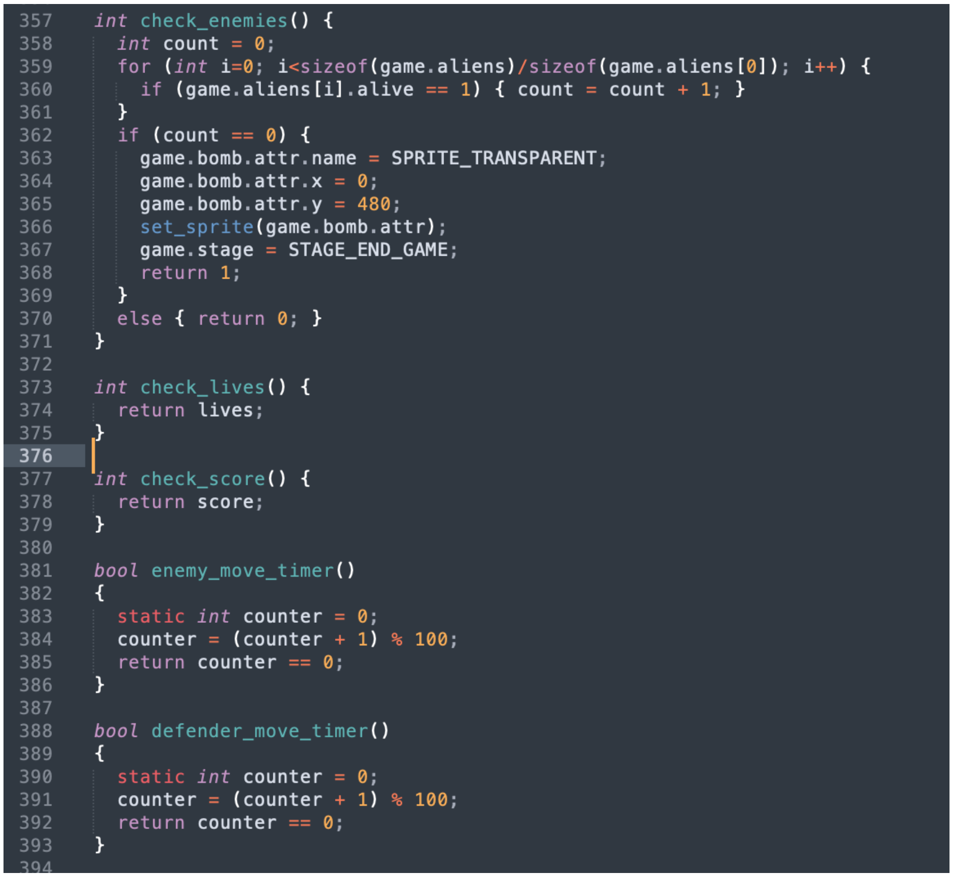
```
void track_bullet() {
  for (int i=0; i<sizeof(game.bullets)/sizeof(game.bullets[0]); i++) {
    // Track alive bullets
    if (game.bullets[i].alive == 1) {
      game.bullets[i].attr.x = game.bullets[i].attr.x;
      game.bullets[i].attr.y = game.bullets[i].attr.y - 4;
      set_sprite(game.bullets[i].attr);
    }
    if (game.bullets[i].attr.y < 25) {
      game.bullets[i].attr.name = SPRITE_TRANSPARENT;
      game.bullets[i].attr.x = 0;
      game.bullets[i].attr.y = 0;
      game.bullets[i].alive = 0;
      set_sprite(game.bullets[i].attr);
    }
  }
}

void drop_bomb(enemy_t *enemy) {
    game.bomb.attr.name = SPRITE_BOMB;
    game.bomb.dir = DIR_DOWN;
    game.bomb.attr.x = enemy->attr.x;
    game.bomb.attr.y = enemy->attr.y + 8;
    set_sprite(game.bomb.attr);
}

void track_bomb() {
  if (dropped == 1) {
    game.bomb.attr.name = SPRITE_BOMB;
    game.bomb.attr.x = game.bomb.attr.x;
    game.bomb.attr.y = game.bomb.attr.y + 2;
    set_sprite(game.bomb.attr);
  }
  if (game.bomb.attr.y >= 440) {
    // Off screen
    game.bomb.attr.name = SPRITE_TRANSPARENT;
    game.bomb.attr.x = 0;
    game.bomb.attr.y = 0;
    set_sprite(game.bomb.attr);
    dropped = 0;
    game.bomb.attr.name = SPRITE_BOMB;
  }
}
```



```c
void check_collision() {
  int right_enemy, left_enemy, bottom_enemy, top_enemy, top_bullet, right_bullet, left_bullet;

  for (int i=0; i<sizeof(game.aliens)/sizeof(game.aliens[0]); i++) {
    right_enemy = game.aliens[i].attr.x + 16;
    left_enemy = game.aliens[i].attr.x - 16;
    bottom_enemy = game.aliens[i].attr.y + 16;
    top_enemy = game.aliens[i].attr.y - 16;

    // Iterate bullets
    pthread_mutex_lock(&game.mu);
    for (int j=0; j<sizeof(game.bullets)/sizeof(game.bullets[0]); j++) {
      if (game.bullets[j].alive == 1) {
        top_bullet = game.bullets[j].attr.y - 8;
        right_bullet = game.bullets[j].attr.x + 2;
        left_bullet = game.bullets[j].attr.x - 2;
        // Check aliens
        if (((right_bullet < right_enemy && right_bullet > left_enemy) &&
             (top_bullet < bottom_enemy && top_bullet > top_enemy)) &&
          game.aliens[i].alive == 1) {
          game.aliens[i].alive = 0;
          game.aliens[i].attr.name = SPRITE_EXPLOSION;
          set_sprite(game.aliens[i].attr);
          usleep(100000);
          game.aliens[i].attr.name = SPRITE_TRANSPARENT;
          set_sprite(game.aliens[i].attr);
          // Reset bullets
          game.bullets[j].attr.name = SPRITE_TRANSPARENT;
          game.bullets[j].attr.x =  0;
          game.bullets[j].attr.y = 0;
          game.bullets[j].alive = 0;
          set_sprite(game.bullets[j].attr);
          // increment score
          score = score + 1;
        }
      }
    }
    pthread_mutex_unlock(&game.mu);
  }
}
```

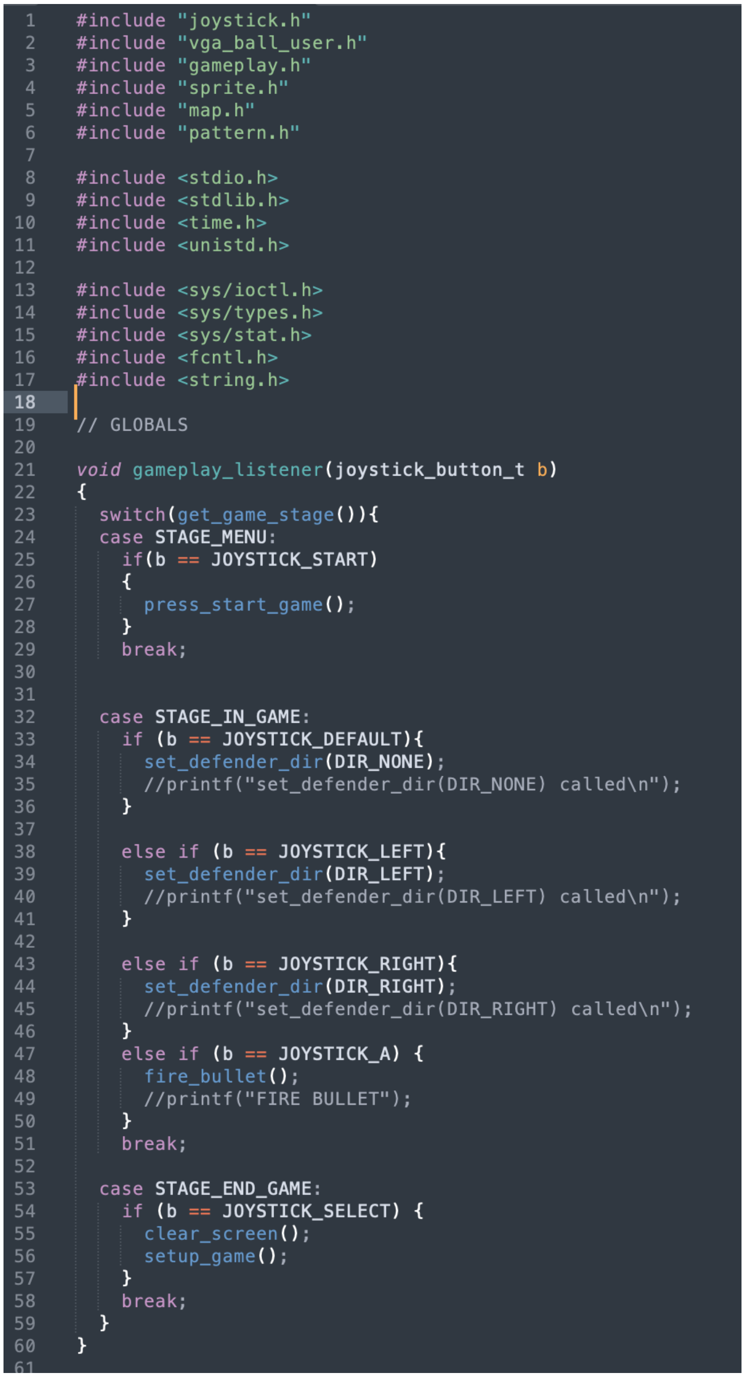


```c
void check_bombs() {
  int right_defender, left_defender, top_defender, bottom_bomb, right_bomb, left_bomb;

  right_defender = game.defender.attr.x + 16;
  left_defender = game.defender.attr.x - 16;
  top_defender = game.defender.attr.y - 16;

  bottom_bomb = game.bomb.attr.y + 8;
  right_bomb = game.bomb.attr.x + 2;
  left_bomb = game.bomb.attr.x - 2;

  if ((right_bomb < right_defender && right_bomb > left_defender) &&
      (bottom_bomb > top_defender)) {
    game.bomb.attr.name = SPRITE_TRANSPARENT;
    game.bomb.attr.x = 0;
    game.bomb.attr.y = 480;
    game.defender.attr.name = SPRITE_EXPLOSION;
    set_sprite(game.defender.attr);
    usleep(100000);
    set_sprite(game.bomb.attr);

    if (lives > 1) {
      lives = lives - 1;
      game.defender.dir0 = DIR_NONE;
      game.defender.attr.i = 0;
      game.defender.attr.y = (MAP_ROW_OFFSET + 13) * 8 + 220;
      game.defender.attr.x = (MAP_COL_OFFSET + 12) * 8;
      game.defender.attr.name = SPRITE_DEFENDER;
      set_sprite(game.defender.attr);
    }
    else { game.stage = STAGE_END_GAME; } // Game over
  }
}
```

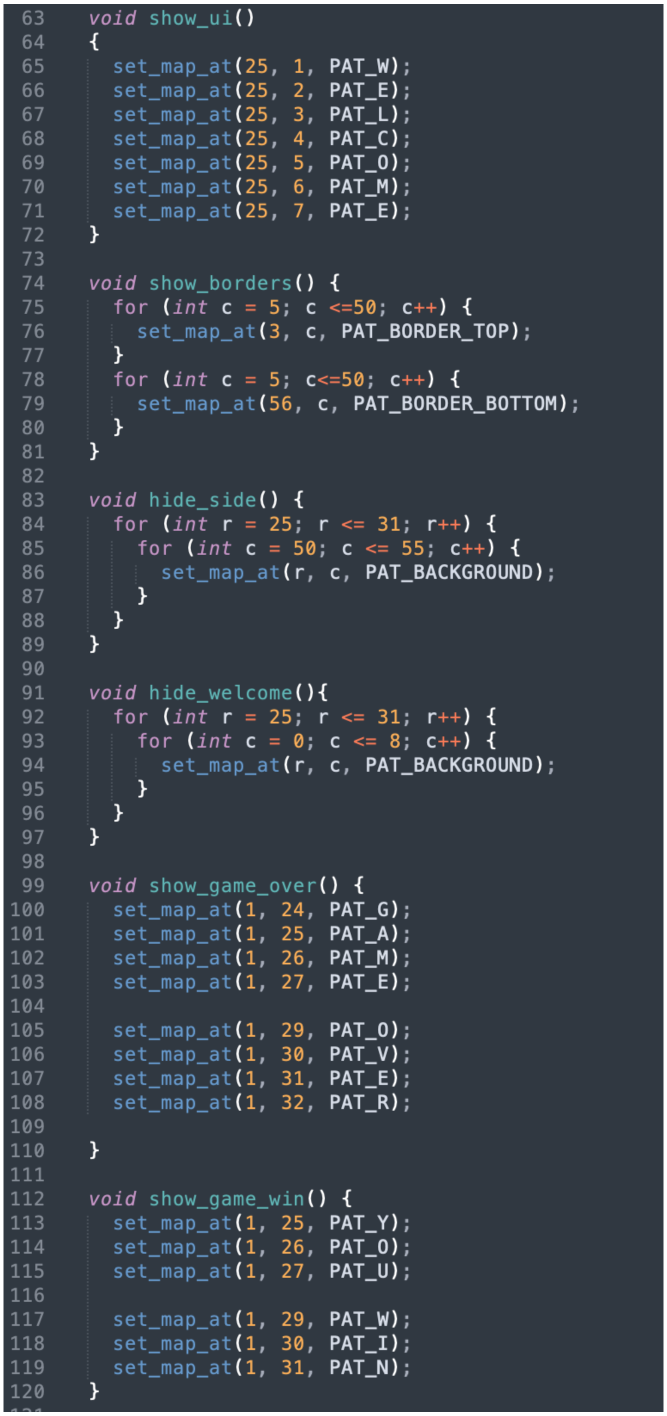


```
int check_enemies() {
  int count = 0;
  for (int i=0; i<sizeof(game.aliens)/sizeof(game.aliens[0]); i++) {
    if (game.aliens[i].alive == 1) { count = count + 1; }
  }
  if (count == 0) {
    game.bomb.attr.name = SPRITE_TRANSPARENT;
    game.bomb.attr.x = 0;
    game.bomb.attr.y = 480;
    set_sprite(game.bomb.attr);
    game.stage = STAGE_END_GAME;
    return 1;
  }
  else { return 0; }
}

int check_lives() {
  return lives;
}

int check_score() {
  return score;
}

bool enemy_move_timer()
{
  static int counter = 0;
  counter = (counter + 1) % 100;
  return counter == 0;
}

bool defender_move_timer()
{
  static int counter = 0;
  counter = (counter + 1) % 100;
  return counter == 0;
}
```

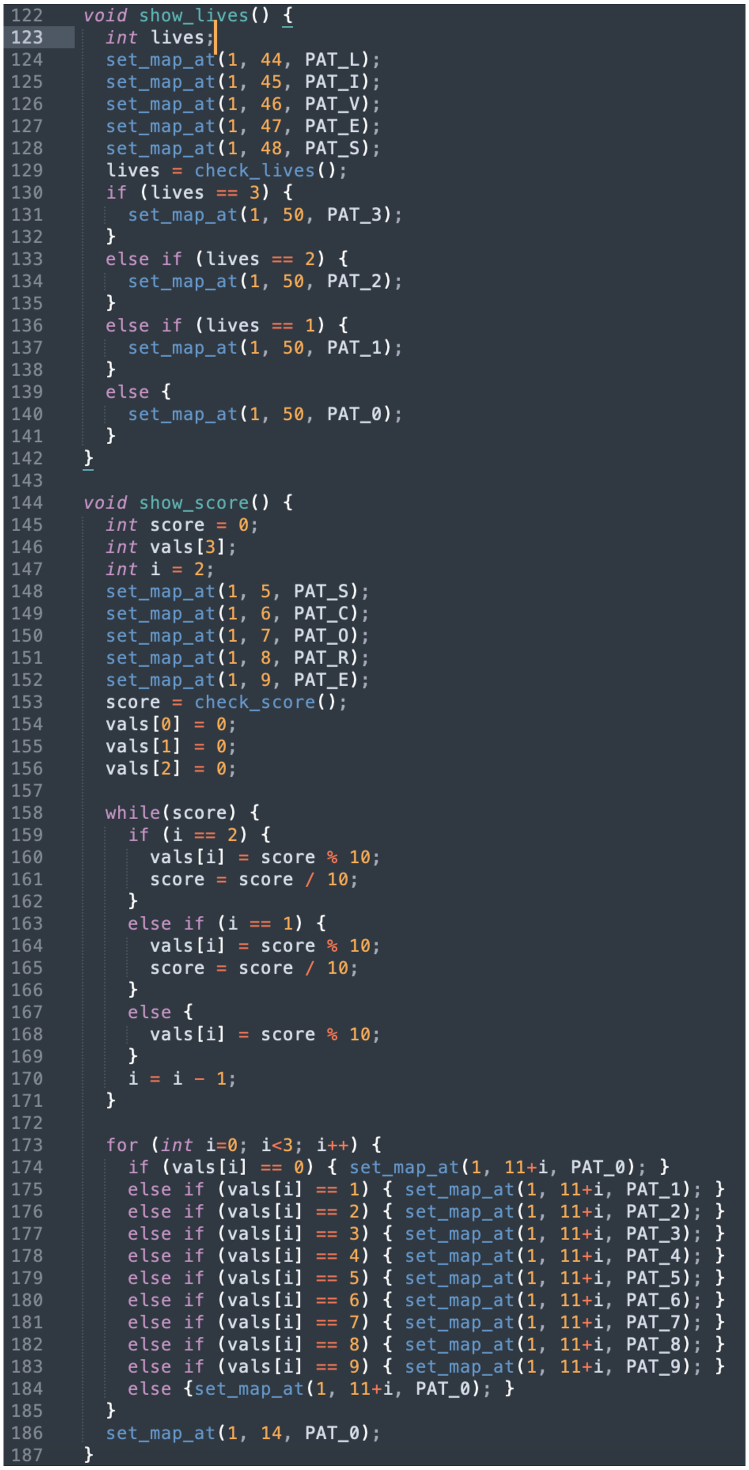
spaceinvaders.c



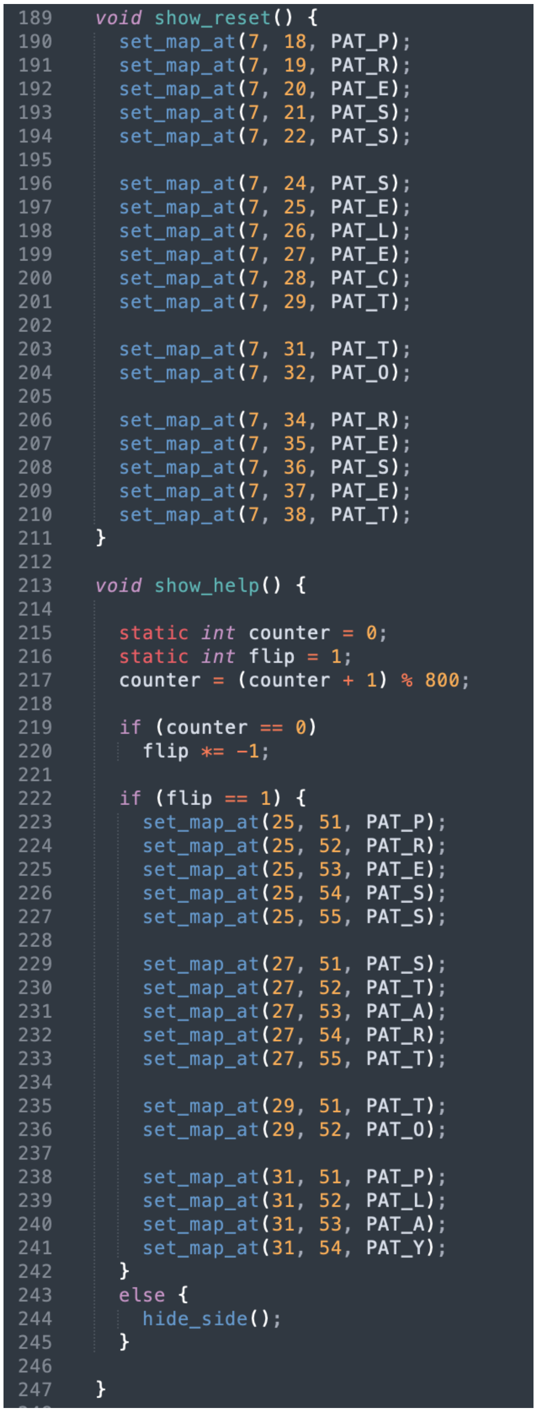
```c
#include "joystick.h"
#include "vga_ball_user.h"
#include "gameplay.h"
#include "sprite.h"
#include "map.h"
#include "pattern.h"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

// GLOBALS

void gameplay_listener(joystick_button_t b)
{
  switch(get_game_stage()){
  case STAGE_MENU:
    if(b == JOYSTICK_START)
    {
      press_start_game();
    }
    break;


  case STAGE_IN_GAME:
    if (b == JOYSTICK_DEFAULT){
      set_defender_dir(DIR_NONE);
      //printf("set_defender_dir(DIR_NONE) called\n");
    }

    else if (b == JOYSTICK_LEFT){
      set_defender_dir(DIR_LEFT);
      //printf("set_defender_dir(DIR_LEFT) called\n");
    }

    else if (b == JOYSTICK_RIGHT){
      set_defender_dir(DIR_RIGHT);
      //printf("set_defender_dir(DIR_RIGHT) called\n");
    }
    else if (b == JOYSTICK_A) {
      fire_bullet();
      //printf("FIRE BULLET");
    }
    break;

  case STAGE_END_GAME:
    if (b == JOYSTICK_SELECT) {
      clear_screen();
      setup_game();
    }
    break;
  }
}

```



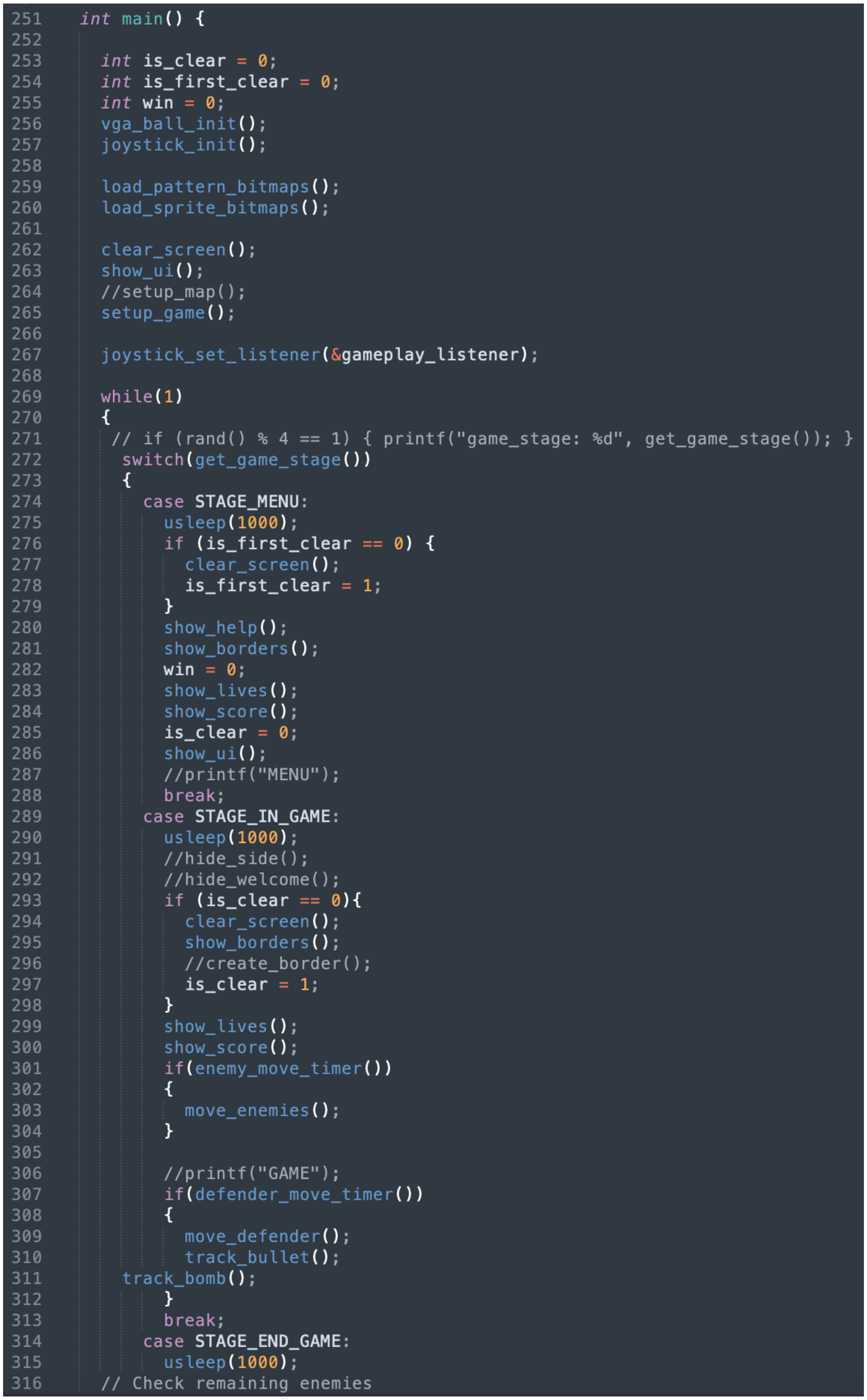
```
void show_ui()
{
  set_map_at(25, 1, PAT_W);
  set_map_at(25, 2, PAT_E);
  set_map_at(25, 3, PAT_L);
  set_map_at(25, 4, PAT_C);
  set_map_at(25, 5, PAT_O);
  set_map_at(25, 6, PAT_M);
  set_map_at(25, 7, PAT_E);
}

void show_borders() {
  for (int c = 5; c <=50; c++) {
    set_map_at(3, c, PAT_BORDER_TOP);
  }
  for (int c = 5; c<=50; c++) {
    set_map_at(56, c, PAT_BORDER_BOTTOM);
  }
}

void hide_side() {
  for (int r = 25; r <= 31; r++) {
    for (int c = 50; c <= 55; c++) {
      set_map_at(r, c, PAT_BACKGROUND);
    }
  }
}

void hide_welcome(){
  for (int r = 25; r <= 31; r++) {
    for (int c = 0; c <= 8; c++) {
      set_map_at(r, c, PAT_BACKGROUND);
    }
  }
}

void show_game_over() {
  set_map_at(1, 24, PAT_G);
  set_map_at(1, 25, PAT_A);
  set_map_at(1, 26, PAT_M);
  set_map_at(1, 27, PAT_E);

  set_map_at(1, 29, PAT_O);
  set_map_at(1, 30, PAT_V);
  set_map_at(1, 31, PAT_E);
  set_map_at(1, 32, PAT_R);

}

void show_game_win() {
  set_map_at(1, 25, PAT_Y);
  set_map_at(1, 26, PAT_O);
  set_map_at(1, 27, PAT_U);

  set_map_at(1, 29, PAT_W);
  set_map_at(1, 30, PAT_I);
  set_map_at(1, 31, PAT_N);
}
```



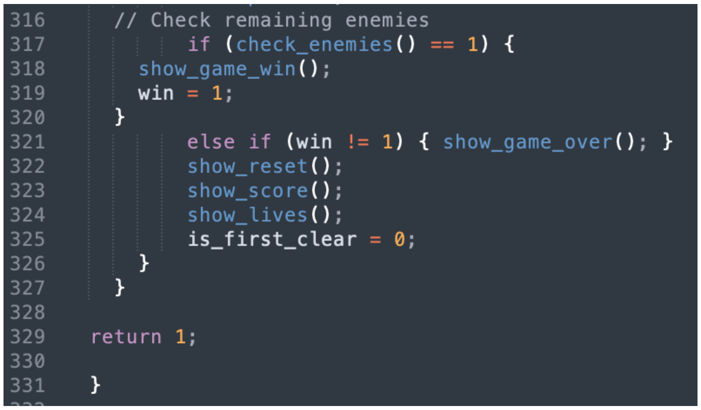
```
void show_lives() {
  int lives;
  set_map_at(1, 44, PAT_L);
  set_map_at(1, 45, PAT_I);
  set_map_at(1, 46, PAT_V);
  set_map_at(1, 47, PAT_E);
  set_map_at(1, 48, PAT_S);
  lives = check_lives();
  if (lives == 3) {
    set_map_at(1, 50, PAT_3);
  }
  else if (lives == 2) {
    set_map_at(1, 50, PAT_2);
  }
  else if (lives == 1) {
    set_map_at(1, 50, PAT_1);
  }
  else {
    set_map_at(1, 50, PAT_0);
  }
}

void show_score() {
  int score = 0;
  int vals[3];
  int i = 2;
  set_map_at(1, 5, PAT_S);
  set_map_at(1, 6, PAT_C);
  set_map_at(1, 7, PAT_O);
  set_map_at(1, 8, PAT_R);
  set_map_at(1, 9, PAT_E);
  score = check_score();
  vals[0] = 0;
  vals[1] = 0;
  vals[2] = 0;

  while(score) {
    if (i == 2) {
      vals[i] = score % 10;
      score = score / 10;
    }
    else if (i == 1) {
      vals[i] = score % 10;
      score = score / 10;
    }
    else {
      vals[i] = score % 10;
    }
    i = i - 1;
  }

  for (int i=0; i<3; i++) {
    if (vals[i] == 0) { set_map_at(1, 11+i, PAT_0); }
    else if (vals[i] == 1) { set_map_at(1, 11+i, PAT_1); }
    else if (vals[i] == 2) { set_map_at(1, 11+i, PAT_2); }
    else if (vals[i] == 3) { set_map_at(1, 11+i, PAT_3); }
    else if (vals[i] == 4) { set_map_at(1, 11+i, PAT_4); }
    else if (vals[i] == 5) { set_map_at(1, 11+i, PAT_5); }
    else if (vals[i] == 6) { set_map_at(1, 11+i, PAT_6); }
    else if (vals[i] == 7) { set_map_at(1, 11+i, PAT_7); }
    else if (vals[i] == 8) { set_map_at(1, 11+i, PAT_8); }
    else if (vals[i] == 9) { set_map_at(1, 11+i, PAT_9); }
    else {set_map_at(1, 11+i, PAT_0); }
  }
  set_map_at(1, 14, PAT_0);
}
```



```
void show_reset() {
  set_map_at(7, 18, PAT_P);
  set_map_at(7, 19, PAT_R);
  set_map_at(7, 20, PAT_E);
  set_map_at(7, 21, PAT_S);
  set_map_at(7, 22, PAT_S);

  set_map_at(7, 24, PAT_S);
  set_map_at(7, 25, PAT_E);
  set_map_at(7, 26, PAT_L);
  set_map_at(7, 27, PAT_E);
  set_map_at(7, 28, PAT_C);
  set_map_at(7, 29, PAT_T);

  set_map_at(7, 31, PAT_T);
  set_map_at(7, 32, PAT_O);

  set_map_at(7, 34, PAT_R);
  set_map_at(7, 35, PAT_E);
  set_map_at(7, 36, PAT_S);
  set_map_at(7, 37, PAT_E);
  set_map_at(7, 38, PAT_T);
}

void show_help() {

  static int counter = 0;
  static int flip = 1;
  counter = (counter + 1) % 800;

  if (counter == 0)
    flip *= -1;

  if (flip == 1) {
    set_map_at(25, 51, PAT_P);
    set_map_at(25, 52, PAT_R);
    set_map_at(25, 53, PAT_E);
    set_map_at(25, 54, PAT_S);
    set_map_at(25, 55, PAT_S);

    set_map_at(27, 51, PAT_S);
    set_map_at(27, 52, PAT_T);
    set_map_at(27, 53, PAT_A);
    set_map_at(27, 54, PAT_R);
    set_map_at(27, 55, PAT_T);

    set_map_at(29, 51, PAT_T);
    set_map_at(29, 52, PAT_O);

    set_map_at(31, 51, PAT_P);
    set_map_at(31, 52, PAT_L);
    set_map_at(31, 53, PAT_A);
    set_map_at(31, 54, PAT_Y);
  }
  else {
    hide_side();
  }

}
```



```c
int main() {

  int is_clear = 0;
  int is_first_clear = 0;
  int win = 0;
  vga_ball_init();
  joystick_init();

  load_pattern_bitmaps();
  load_sprite_bitmaps();

  clear_screen();
  show_ui();
  //setup_map();
  setup_game();

  joystick_set_listener(&gameplay_listener);

  while(1)
  {
   // if (rand() % 4 == 1) { printf("game_stage: %d", get_game_stage()); }
    switch(get_game_stage())
    {
      case STAGE_MENU:
        usleep(1000);
        if (is_first_clear == 0) {
          clear_screen();
          is_first_clear = 1;
        }
        show_help();
        show_borders();
        win = 0;
        show_lives();
        show_score();
        is_clear = 0;
        show_ui();
        //printf("MENU");
        break;
      case STAGE_IN_GAME:
        usleep(1000);
        //hide_side();
        //hide_welcome();
        if (is_clear == 0){
          clear_screen();
          show_borders();
          //create_border();
          is_clear = 1;
        }
        show_lives();
        show_score();
        if(enemy_move_timer())
        {
          move_enemies();
        }

        //printf("GAME");
        if(defender_move_timer())
        {
          move_defender();
          track_bullet();
    track_bomb();
        }
        break;
      case STAGE_END_GAME:
        usleep(1000);
  // Check remaining enemies
```



```
  // Check remaining enemies
        if (check_enemies() == 1) {
    show_game_win();
    win = 1;
  }
        else if (win != 1) { show_game_over(); }
        show_reset();
        show_score();
        show_lives();
        is_first_clear = 0;
    }
  }

return 1;

}
```