Arcade Poker Deck Builder Game: Balatro Minus

CSEE 4840: Embedded Systems Design

Spring 2025 Final Report

Team Members (UNI):
Mahdi Ali-Raihan (mma2268), Timothy Melendez (tjm2196),
Mario Carrillo-Bello (mc5132), Julio Ramirez (jar2358)

Contents

Introduction

The goal of this project was to design and implement an arcade version of poker, inspired by the roguelike card game *Balatro*. In this game, a hand of eight cards is dealt, from which the player may choose to play a poker hand or discard and draw. The task is to reach the required target score for each blind. As the user progresses through the game, the target score increases. To meet the target score, the user must form strong poker hands to earn more points. After beating the target score, the user has a probability of earning a random 'Joker' card, which will help the user defeat much more difficult target scores down the road. The game is displayed on a 640 x 480 VGA monitor created by tile generator hardware on the FPGA and is controlled via an NES controller.

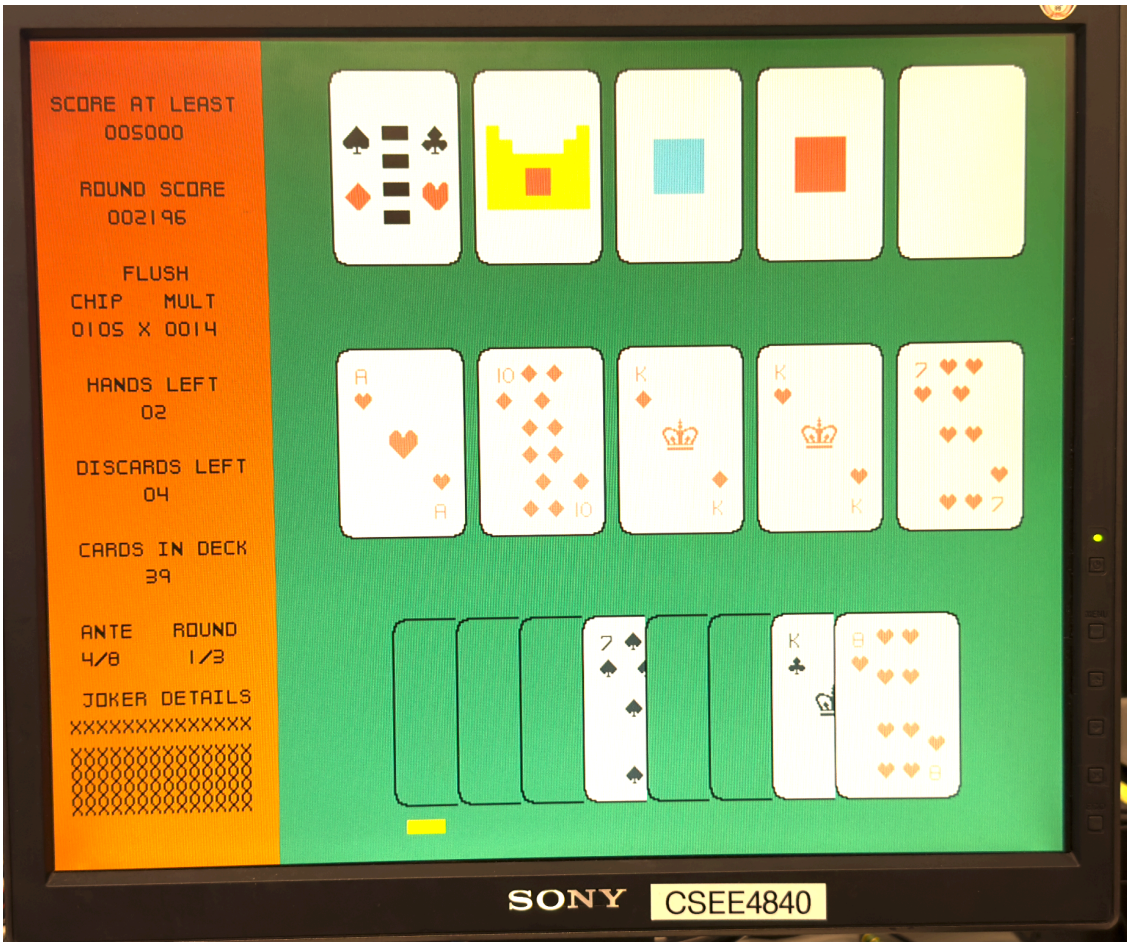


Figure 1: Image of Final Game Display.

Figure 2: Image of Starting Screen

System Overview

The figure below provides an overview of the implementation of our system for Balatro Minus. The game makes use of the FPGA's VGA hardware module to send display data to a VGA monitor, as well as the USB networking protocol to receive input data from an NES controller to interact with different game components. A VGA driver module is used to communicate and write to the specified address, which is then processed by the VGA module to correctly write either to the Tileset RAM, Tilemap RAM, or Palette RAM. The final output of the pipeline is a 4-bit color output that is decoded to a 24-bit RGB color (byte per color).

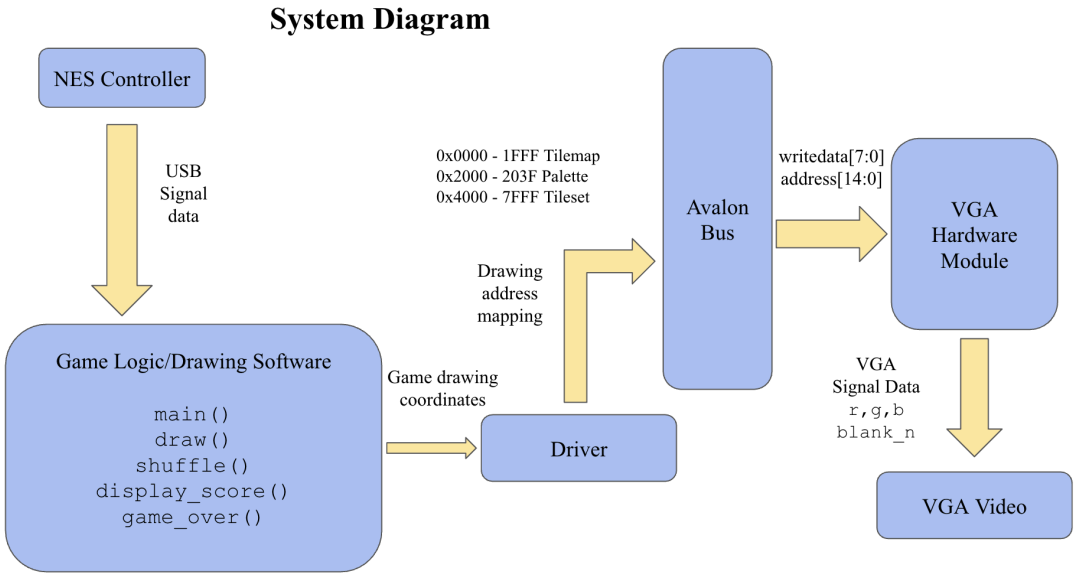


Figure 3: Block diagram of the overall system.

Hardware Design

VGA Video Rendering

The hardware for the game relies on tile-based graphics to display all components, including poker cards for gameplay and text for displaying information about the player's score and the status of the game. We use a VGA monitor with a 640x480 resolution. The chosen tile size on the screen is 8x8 bits, resulting in a tile grid of 80x60 tiles. The tileset consists of 256 unique tiles used to create each component displayed on the screen, ranging from the edges to outline a single card to each of the four suites in the deck. Each pixel on the display takes up 4 bits of memory, for a total memory consumption of 0.1536 MB, which is far below the total memory of ~0.5 MB.

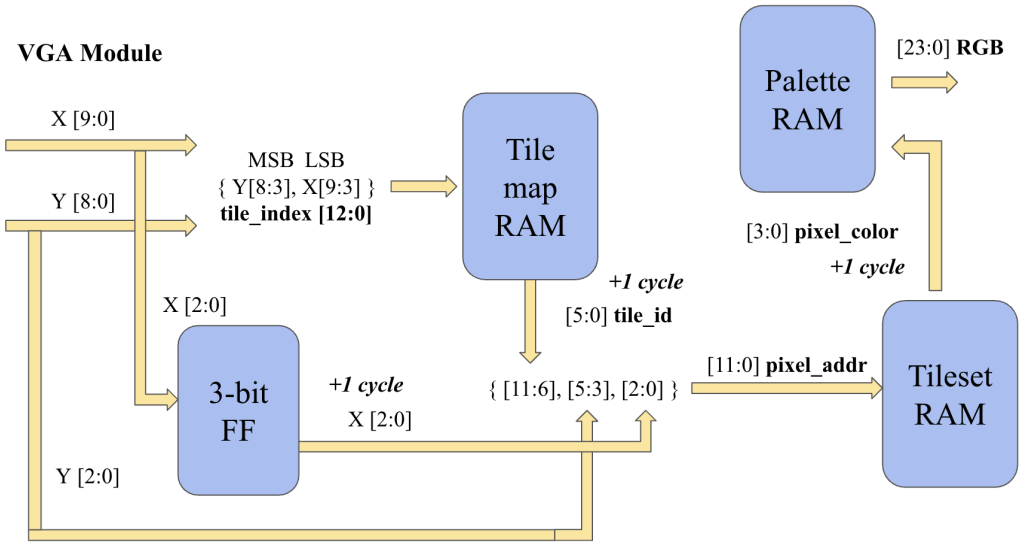


Figure 4: Block diagram of the original VGA module design used to generate the tile-based graphics (8x8 tiles, 64 tile_ids, 16 unique palette colors that give 24-bit RGB).

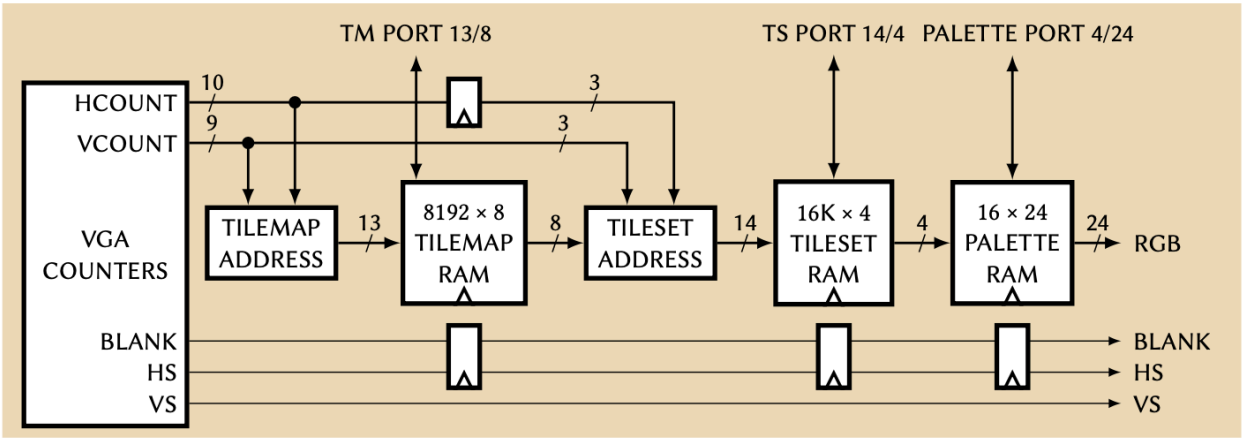


Figure 5: Block diagram of the actualized VGA module pipeline design to generate the tile-based graphics (Credit: Stephen A. Edwards).

Given the 15-bit address size, decoding which memory to write to involves decoding the MSB of the address to determine if the data is written to the tileset memory; if not, then the next MSB is used to determine if the palette memory is written to. If neither is active, then the tilemap memory is written to. An extra 3-bit register flip-flop was used to keep the hcount[2:0] in sync with the rest of the pipeline due to a mismatch in pixel row and columns. This complete pipeline can be customized to suit larger tilesets (increasing amount of address bits) or increasing the amount of unique colors (increasing color index per pixel), etc.

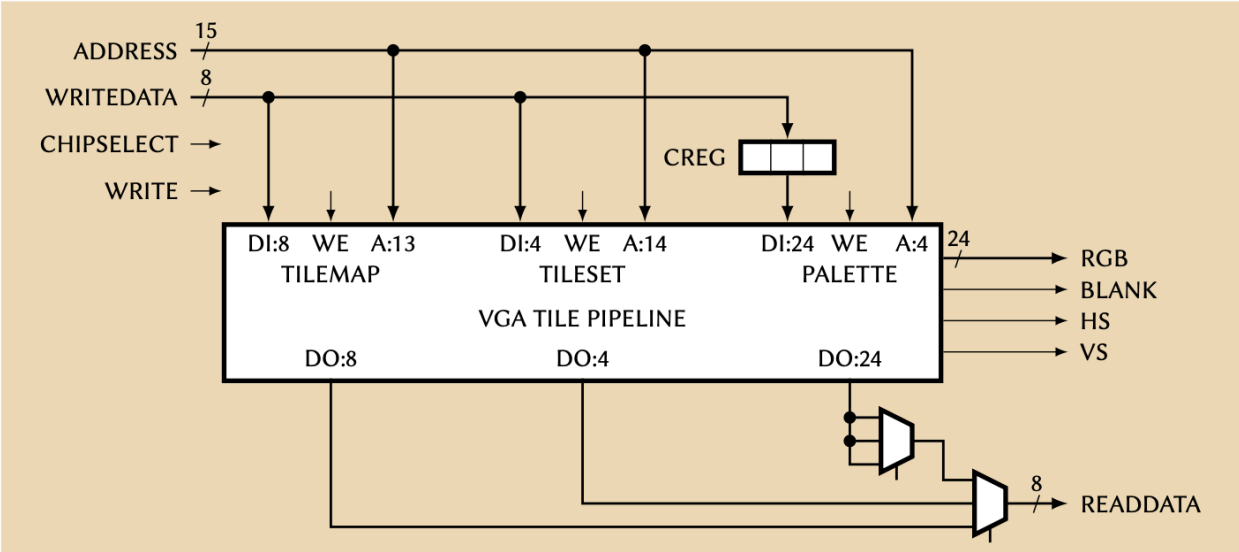


Figure 6: Block diagram of the actualized and complete VGA module to generate the tile-based graphics (Credit: Stephen A. Edwards).

The complete, detailed design and implementation of the aforementioned VGA tile generator was created by Stephen A. Edwards, in which extensive documentation is provided https://www.cs.columbia.edu/~sedwards/classes/2025/4840-spring/tiles.pdf.

Credit is given to Professor Edwards for allowing us to use his design of the tile generation pipeline.

Software Design

The software for the game consists of several components: the kernel device driver for interfacing with the Avalon bus to write to specific memory addresses, the userspace library to communicate with the device driver using ioctl calls, USB protocol controller software, and the game logic itself.

VGA Device Driver

The device driver for Balatro Minus utilizes three register allocations: *TILEMAP_BASE (address 0x0)*, *PALETTE_BASE (address 0x2000)*, and *TILESET_BASE(address 0x4000)*. Several functions were also used to go from individual pixels to tiles to generate the various game components on the screen. *set_pixels* is used to set the color of individual pixels based on their coordinates on the VGA monitor, *set_palette* determines the palette of possible colors for assignment to each pixel, and *set_tile* assigns tiles using a coordinate system of rows and columns on the VGA monitor, assigning an ID to each tile.

Userspace Program

The userspace program uses ioctl calls to communicate with the VGA device driver to load the graphics for the game. It loads in the palette, tileset, and tilemap using the associated .hex files, subsequently drawing cards for the user's hand using the loaded tiles. It then runs through several functions to draw other essential graphic components of the game, such as the target score, the user's current score, the number of hands and discards remaining for the round, the number of cards in the deck, the ante number, and the round number. It also runs functions for the user's cursor on the screen and additional tile graphics when selecting or deselecting a card, as well as drawing selected cards to the center row of the display.

Game Logic

The following algorithms outline the various functionalities that are implemented in the game.

/* Hand Evaluation Functions */

- **HandValue get_hand_value(enum HandType hand_type);** - Returns the chip value and multiplier for a given hand type
- **int check_straight(int rank_counts);** - Checks if the selected cards form a straight
- **enum HandType evaluate_selected_cards();** - Evaluates the currently selected cards and returns the hand type
- **enum HandType evaluate_hand(Card hand, int num_cards);** - Evaluates any set of cards and returns the hand type

/* Card and Deck Management Functions */

- **void init_deck();** - Initializes the deck of 52 cards
- **void shuffle_deck();** - Shuffles the deck using the Fisher-Yates algorithm
- **Card draw_card();** - Draws the topmost card from the deck
- **int cards_remaining();** - Returns the number of cards remaining in the deck
- **void draw_initial_pool_of_cards();** - Draws eight initial cards for the player's pool
- **void draw_replacement_cards();** - Replaces played or discarded cards with new ones
- **uint8_t drawn_to_index();** - Maps drawn cards to tile indices for display

/* Game State Management Functions */

- **void init_antes();** - Initializes the ante values and blinds
- **void init_jokers();** - Initializes all joker cards to inactive
- **int play_selected_hand();** - Plays the currently selected cards and returns the score
- **void discard_selected_cards();** - Discards the currently selected cards
- **int check_win_condition(int player_score);** - Checks if the player has reached the target score

- **void advance_game_state();** - Advances to the next blind or ante after winning
- **void round_reset();** - Resets the game state for a new round
- **void hard_reset();** - Performs a complete reset of the game
- **void game_over();** - Handles game over state when player loses
- **void game_won();** - Handles winning the entire game (after beating Ante 8)
- **void get_current_blind_info(char blind_name, int target_score);** - Gets information about the current blind
- **void game_loop_vga();** - Main game loop that handles rendering and input

/* VGA Display Functions */

- **const char get_hand_name_vga(enum HandType hand);** - Returns the string name of a hand type
- **char get_target_score_vga(int target_score);** - Formats target score for display
- **char get_round_score_vga(int round_score);** - Formats round score for display
- **char get_chip_vga(int chip_value);** - Formats chip value for display
- **char get_mult_vga(int multiplier);** - Formats multiplier for display
- **char get_hands_left_vga();** - Returns a formatted string of hands remaining
- **char get_discards_left_vga();** - Returns formatted string of discards remaining
- **char get_cards_in_deck_vga();** - Returns formatted string of cards in deck
- **char get_ante_vga();** - Returns a formatted string of the current ante
- **char get_round_vga();** - Returns a formatted string of the current round
- **uint8_t get_cursor_position();** - Returns the current cursor position
- **uint8_t get_clear_cursor_position();** - Returns the previous cursor position
- **void update_clear_cursor_position();** - Updates the clear cursor position
- **uint8_t get_selected_cards_array();** - Returns array indicating which cards are selected

/* Controller Input Functions */

- **void initialize_debounce();** - Initializes the debounce timer
- **int check_controller_input(unsigned char report);** - Interprets controller input report
- **void toggle_card_selection();** - Toggles selection state of card at cursor
- **void move_cursor(int direction);** - Moves cursor left or right
- **void process_controller_input(unsigned char report);** - Processes controller input

/* Score Calculation Functions */

- **int get_individual_card_chip(Card card);** - Returns chip value for a single card
- **int calculate_hand_type_specific_chips();** - Calculates chips based on the hand type
- **int calculate_played_hand_chips(Card hand, int num_cards, enum HandType hand_type);** - Calculates chip values for a played hand

/* Joker Effect Functions */

- **int apply_smiley_face_joker(Card hand, int num_cards, int base_multiplier);** - Adds 5 multiplier per face card
- **int apply_even_steven_joker(Card hand, int num_cards, int base_multiplier);** - Adds 4 multiplier per even rank card
- **int apply_odd_todd_joker(Card hand, int num_cards, int base_chips);** - Adds 31 chips per odd rank card
- **Card transform_card_suit_bundler(Card card);** - Transforms card suit for Suit Bundler joker
- **void apply_suit_bundler_joker(Card hand, int num_cards, Card transformed_hand);** - Makes spades equal clubs and diamonds equal hearts
- **void activate_the_one_joker();** - Activates The One joker effect
- **int check_the_one_joker();** - Checks if The One joker is active
- **int apply_blue_dot_joker(int base_chips);** - Adds 50 chips to hand
- **int apply_red_dot_joker(int base_multiplier);** - Adds 10 to multiplier
- **int apply_green_check_joker(int target_score, int current_score, int hands_left);** - Reduces target score to 25%
- **void apply_stevie_dott_joker(Card *hand, int num_cards, int multiplier, int card_chips);** - Changes multiplier or doubles chips based on rank sum
- **void apply_all_joker_effects(Card *hand, int num_cards, enum HandType *hand_type, int chips, int multiplier, int target_score, int current_score, int hands_left);** - Applies all active joker effects

/* Joker Management Functions */

- **void joker_drop();** - Determines if a joker drops after beating a blind/round
- **int count_active_jokers(Card joker_cards[10]);** - Counts how many active jokers the player has

Hardware-Software Interface

Driver

Our game utilized three main interface calls to communicate to the specified address range and memories of our tiles, pixel colors, and map.

`set_pixels()` allowed us to write our pixel colors of a unique tile_id:

```
static void set_pixels(vga_poker_pixels_t *pixel_coors)
{
   int tile_id;
   int i;
   tile_id = pixel_coors->tile_id;
   for (i = 0; i < 64; i++) {
       int color;
       int new_addr;
       color = pixel_coors->pixels[i];
       new_addr = TILESET_BASE + (tile_id << 6) + i;
       iowrite8(color, dev.virtbase + new_addr);
   }
}
```

`set_palette()` allowed us to write the actual colors we wanted in our memory:

```
static void set_palette(vga_poker_palette_t *palette)
{
   int base;

   if (palette->index >= 16) return;

   base = PALETTE_BASE + (palette->index << 2);
   /* write into creg[7:0], [15:8], [23:16] */
   iowrite8(palette->red,   dev.virtbase + base + 0);
   iowrite8(palette->green, dev.virtbase + base + 1);
   iowrite8(palette->blue,  dev.virtbase + base + 2);
   /* commit into palette[pl->index] */
   iowrite8(0, dev.virtbase + base + 3);
}
```

`set_tile()` allowed us to write our tile to specific region on the screen:

```
static void set_tile(vga_poker_tile_coords_t *tile_coords)
{
   int row;
   int col;
   int tile_id;
   unsigned short new_tile_addr;
```

```
    row = tile_coords->row;
    col = tile_coords->col;
    tile_id = tile_coords->tile_id;
    new_tile_addr =  TILEMAP_BASE + (row << 7 | col);
    iowrite8(tile_id, dev.virtbase + new_tile_addr);

}
```

Each of these interface calls were wrapped around our ioctl calls to from userspace and utilized in `poker.h` for drawing dynamic tiles to the screen.

Controller

Our game utilizes the Nintendo NES (which in reality is a DragonRise Inc. USB Gamepad) controller to access in-game controls, which interface with the DE1-SoC board via USB. The interface uses the libusb C library to communicate using the USB networking protocol. We used the Lab 2 base code for the controller, but modified it in such a way that it allowed us to use the provided controller.

Figure 7: Image of the NES controller used for the project.

Button Functionality

The controller buttons serve the following purposes in our poker game:

- **D-Pad Left/Right**: Navigates the cursor between cards in the player's hand
- **A Button**: Selects the card at the current cursor position
- **B Button**: Removes the card at the cursor position from the current selection
- **Start Button**: Plays the currently selected hand (1-5 cards allowed)
- **Select Button**: Discards the currently selected cards and draws replacements (limited uses per round)

Each button press is processed with debounce protection to prevent unintentional repeated inputs. The game also manages button state transitions to ensure actions are only performed once per button press.

Controller Input Encoding

The controller communicates via an 8-byte report format. Each button press generates a unique pattern of bytes that our code interprets:

Button	Byte Pattern (Hex)
None (No Press)	01 7F 7F 7F 7F 0F 00 00
Left	01 7F 7F 00 7F 0F 00 00
Right	01 7F 7F FF 7F 0F 00 00
Up	01 7F 7F 7F 00 0F 00 00
Down	01 7F 7F 7F FF 0F 00 00
A	01 7F 7F 7F 7F 2F 00 00
B	01 7F 7F 7F 7F 4F 00 00
Select	01 7F 7F 7F 7F 0F 10 00
Start	01 7F 7F 7F 7F 0F 20 00

VGA Interface

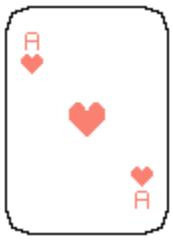
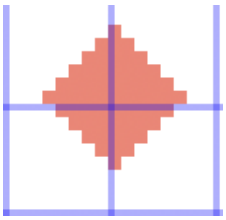
For the VGA interface, we used a 15-bit-wide address bus, with up to 8 bits of data writing. The table below shows the address mapping.

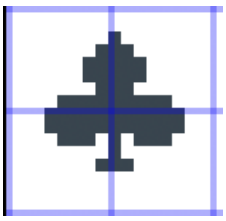
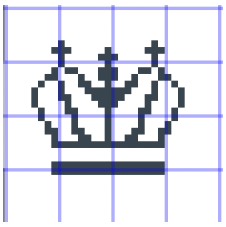
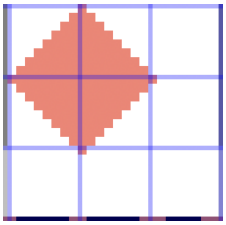
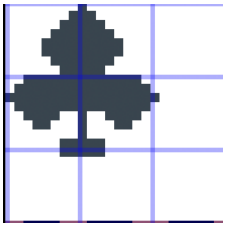
Address Range (15 Bits)	8-Bit Write Bus	Information
0x0 - 0x1FFF *The address offset is calculated as follows:*	[7:0] tile_id	Tilemap addresses, 8K range of memory where each id is 8 bits

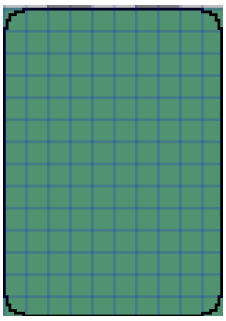
BASE_ADDR + row << 7 | col		
0x2000 - 0x203F *The first 3 bytes written to an address registers the color in the color register, and the 4th byte writes that color to the screen.*	[7:0] pixel_red [7:0] pixel_green [7:0] pixel_blue [7:0] 0x0	Palette color addresses, 64 B of colors, each color is 4 bytes
0x4000 - 0x7FFF *The address offset is calculated as follows where i is the local pixel value of that tile:* *BASE_ADDR + tile_id << 6 | i*	[7:0] pixel_color	Tileset addresses 16K range where each color output is represent as 4 bits

Resource Allocation

The following table outlines the memory allocation for various categories implemented in Balatro Minus. It should be noted that each category may be made up of more than one 8x8 tile.

Category	Tiles	Size (Bits)	# of Sprites	Total Size (Bits)
Playing Card (Ace, 2-10, Jack, Queen, King)		80 x 112 x 4	52	80 * 112 * 4 * 52 = 1,863,680
Heart (small)		16 x 16 x 4	1	16 * 16 * 4 * 1 = 1024
Spade (small)		16 x 16 x 4	1	16 * 16 * 4 * 1 = 1024
Diamond (small)		16 x 16 x 4	1	16 * 16 * 4 * 1 = 1024

Club (small)		16 x 16 x 4	1	16 * 16 * 4 * 1 = 1024
Jester Face		40 x 40 x 4	1	40 * 40 * 4 = 6,400
King Face		24 x 24 x 4	1	24 * 24 * 4 = 2,304
Queen Face		32 x 32 x 4	1	32 * 32 * 4 = 4,096
Heart (Big)		24 x 24 x 4	1	24 * 24 * 4 = 2,304
Spade (Big)		24 x 24 x 4	1	24 * 24 * 4 = 2,304
Diamond (Big)		24 x 24 x 4	1	24 * 24 * 4 = 2,304
Club (Big)		24 x 24 x 4	1	24 * 24 * 4 = 2,304

Joker Card		80 x 112 x 4	10	80 * 112 * 4 * 10 = 358,400
Empty Card		80 x 112 x 4	1	80 * 112 * 4 * 1 = 35,840
Letters		8 x 8 x 4	26	8 * 8 * 4 * 26 = 6,656
Numbers (0-9)		8 x 8 x 4	10	8 * 8 * 4 * 10 = 2,560
			TOTAL	2,293,248

Closing

Task Allocation

Timothy Melendez: Hardware initialization, Driver, Drawing software library
Mahdi Ali-Raihan: Pixel Art, Tile Set, Tile Map
Mario Carrillo-Bello: Game Logic and Controller
Julio Ramirez: Hardware, Driver, Game Logic, Controller

Challenges and Lessons Learned

Overall, this project was very insightful and challenging. The hardware-software interface is the key component of a successful embedded system project. We learned that hardware design must be carefully and thought through in order to prevent unwanted debugging and faulty logic. Our

team ended up relying on professor Edwards's tile generator design to test and implement our device drivers, and user space programs. If we had started our design as soon as we received a design review, we would have more time to implement our unique hardware design. Relying on teammates who specialized on certain tasks of the projects was also very useful to divide and conquer. This project taught us a lot about the intricacies of hardware design and software implementation which is crucial to any embedded system. Plus, it was fun to make a game :)

References and Source Code

vga_poker.c

```C/C++
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_poker.h"
#define DRIVER_NAME "vga_poker"
/* Device registers */
#define TILEMAP_BASE   0x0000  /* 8 KiB: 0x0000– addressing */
#define PALETTE_BASE   0x2000  /*  64 B:  0x2000–0x203F */
#define TILESET_BASE   0x4000  /* 16 KiB: 0x4000–0x7FFF */

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

static void set_tile(vga_poker_tile_coords_t *tile_coords)
{
```

```
    int row;
    int col;
    int tile_id;
    unsigned short new_tile_addr;
    row = tile_coords->row;
    col = tile_coords->col;
    tile_id = tile_coords->tile_id;
    new_tile_addr =  TILEMAP_BASE + (row << 7 | col);
    iowrite8(tile_id, dev.virtbase + new_tile_addr);

}

static void set_pixels(vga_poker_pixels_t *pixel_coors)
{
    int tile_id;
    int i;
    tile_id = pixel_coors->tile_id;
    for (i = 0; i < 64; i++) {
        int color;
        int new_addr;
        color = pixel_coors->pixels[i];
        new_addr = TILESET_BASE + (tile_id << 6) + i;
        iowrite8(color, dev.virtbase + new_addr);
    }
}

static void set_palette(vga_poker_palette_t *palette)
{
    int base;

    if (palette->index >= 16) return;

    base = PALETTE_BASE + (palette->index << 2);
    /* write into creg[7:0], [15:8], [23:16] */
    iowrite8(palette->red,   dev.virtbase + base + 0);
    iowrite8(palette->green, dev.virtbase + base + 1);
    iowrite8(palette->blue,  dev.virtbase + base + 2);
    /* commit into palette[pl->index] */
    iowrite8(0, dev.virtbase + base + 3);
}
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
```

```
 */
static long vga_poker_ioctl(struct file *f, unsigned int cmd, unsigned long
arg)
{
    vga_poker_arg_t vla;
    switch (cmd) {

   case VGA_POKER_SET_TILE:
        if (copy_from_user(&vla, (vga_poker_arg_t *) arg,
                   sizeof(vga_poker_arg_t)))
            return -EACCES;
        set_tile(&vla.tile_coords);
        break;
   case VGA_POKER_SET_PALETTE:
        if (copy_from_user(&vla, (vga_poker_arg_t *) arg,
                   sizeof(vga_poker_arg_t)))
            return -EACCES;
        set_palette(&vla.palette_t);
        break;
   case VGA_POKER_SET_PIXELS:
        if (copy_from_user(&vla, (vga_poker_arg_t *) arg,
                   sizeof(vga_poker_arg_t)))
            return -EACCES;
        set_pixels(&vla.pixel_coors);
        break;
    default:
        return -EINVAL;
    }
    return 0;
}
/* The operations our device knows how to do */
static const struct file_operations vga_poker_fops = {
    .owner     = THIS_MODULE,
    .unlocked_ioctl = vga_poker_ioctl,
};
/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_poker_misc_device = {
    .minor     = MISC_DYNAMIC_MINOR,
    .name      = DRIVER_NAME,
    .fops      = &vga_poker_fops,
};
/*
 * Initialization code: get resources (registers) and display
 * a welcome message
```

```
 */
static int __init vga_poker_probe(struct platform_device *pdev)
{
    int ret;
    /* Register ourselves as a misc device: creates /dev/vga_poker */
    ret = misc_register(&vga_poker_misc_device);
    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }
    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }
    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }
    return 0;
out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&vga_poker_misc_device);
    return ret;
}
/* Clean-up code: release resources */
static int vga_poker_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&vga_poker_misc_device);
    return 0;
}
/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_poker_of_match[] = {
    { .compatible = "csee4840,vga_tiles-1.0" },
    {},
```

```
};
MODULE_DEVICE_TABLE(of, vga_poker_of_match);
#endif
/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_poker_driver = {
    .driver    = {
        .name  = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_poker_of_match),
    },
    .remove    = __exit_p(vga_poker_remove),
};
/* Called when the module is loaded: set things up */
static int __init vga_poker_init(void)
{
   pr_info(DRIVER_NAME ": init\n");
   return platform_driver_probe(&vga_poker_driver, vga_poker_probe);
}
/* Calball when the module is unloaded: release resources */
static void __exit vga_poker_exit(void)
{
   platform_driver_unregister(&vga_poker_driver);
   pr_info(DRIVER_NAME ": exit\n");
}
module_init(vga_poker_init);
module_exit(vga_poker_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");
```

vga_poker.h

```
C/C++
#ifndef _VGA_POKER_H
#define _VGA_POKER_H

#include <linux/ioctl.h>

typedef struct {
 short row, col;
 char tile_id;
} vga_poker_tile_coords_t;
```

```
typedef struct {
 char tile_id;
 char pixels[64];
} vga_poker_pixels_t;

typedef struct {
 char index, red, green, blue;
} vga_poker_palette_t;

typedef struct {
 vga_poker_tile_coords_t tile_coords;
 vga_poker_pixels_t pixel_coors;
 vga_poker_palette_t palette_t;
} vga_poker_arg_t;


#define VGA_POKER_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_POKER_SET_TILE  _IOW(VGA_POKER_MAGIC, 1, vga_poker_arg_t)
#define VGA_POKER_SET_PALETTE  _IOW(VGA_POKER_MAGIC, 2, vga_poker_arg_t)
#define VGA_POKER_SET_PIXELS  _IOW(VGA_POKER_MAGIC, 3, vga_poker_arg_t)

#endif
```

poker.h

```
C/C++
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>
#include "vga_poker.h"

/* Defined dynamic tile rows and cols to update */
#define TARGET_SCORE_ROW 6
#define ROUND_SCORE_ROW 12
```

```
#define SCORE_DIGITS 6
#define SCORE_COL_START 6
#define SCORE_COL_END 11

#define HAND_TYPE_ROW 16
#define HAND_TYPE_COL_START 2
#define HAND_TYPE_COL_END 16

#define CHIP_MULT_CALC_ROW 20
#define CHIP_COL_START 3
#define CHIP_COL_END 6
#define MULT_COL_START 10
#define MULT_COL_END 13

#define HANDS_LEFT_ROW 26
#define HANDS_LEFT_COL 9

#define DISCARD_ROW 32
#define DISCARD_COL 9

#define CARDS_IN_DECK_ROW 38
#define CARDS_IN_DECK_COL 8 /* ends at 9*/

#define ANTE_ROUND_ROW 44
#define ANTE_COL 3
#define ROUND_COL 11

#define JOKER_NAME_ROW 49
#define JOKER_INFO_ROW_START 51
#define JOKER_INFO_ROW_END 55
#define JOKER_NAME_COL_START 2
#define JOKER_NAME_COL_END 15

#define JOKER_ROW 2
#define JOKER_1_COL 23
#define JOKER_2_COL 34
#define JOKER_3_COL 45
#define JOKER_4_COL 56
#define JOKER_5_COL 67

#define CURSOR_ROW 57
#define CURSOR_COL 27
#define SELECTED_ROW 40
#define SELECTED_COL 27
```

```
#define PLAYED_ROW 22
#define PLAYED_1_COL 23
#define PLAYED_2_COL 34
#define PLAYED_3_COL 45
#define PLAYED_4_COL 56
#define PLAYED_5_COL 67

#define HAND_ROW_START 42
#define HAND_ROW_END 55
#define HAND_1_COL_START 27
#define HAND_1_COL_END 31
#define HAND_2_COL_START 32
#define HAND_2_COL_END 36
#define HAND_3_COL_START 37
#define HAND_3_COL_END 41
#define HAND_4_COL_START 42
#define HAND_4_COL_END 46
#define HAND_5_COL_START 47
#define HAND_5_COL_END 51
#define HAND_6_COL_START 52
#define HAND_6_COL_END 56
#define HAND_7_COL_START 57
#define HAND_7_COL_END 61
#define HAND_8_COL_START 62
#define HAND_8_COL_END 71

#define CARD_TILE_ID_ROWS 14
#define CARD_TILE_ID_COLS 10

int vga_poker_fd;
/* row col */
int tile_map[60][80];
/* C, D, S H*/
uint8_t deck[52][CARD_TILE_ID_ROWS][CARD_TILE_ID_COLS];
uint8_t jokers[10][CARD_TILE_ID_ROWS][CARD_TILE_ID_COLS];
uint8_t card[CARD_TILE_ID_ROWS][CARD_TILE_ID_COLS];

uint8_t card_slot[CARD_TILE_ID_ROWS][CARD_TILE_ID_COLS] = {
 { 0xEE, 0xF8, 0xF8, 0xF8, 0xF8, 0xF8, 0xF8, 0xF8, 0xF8, 0xEF },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
```

```
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF9, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0x5F, 0xFA },
 { 0xF0, 0xFB, 0xFB, 0xFB, 0xFB, 0xFB, 0xFB, 0xFB, 0xFB, 0xF1 }
};

void set_palette(const vga_poker_palette_t *arg)
{
 vga_poker_arg_t vga;
 vga.palette_t = *arg;

 if (ioctl(vga_poker_fd, VGA_POKER_SET_PALETTE, &vga)) {
   perror("ioctl(vga_poker_SET_BACKGROUND) failed");
   return;
 }
}

void set_pixels(const vga_poker_pixels_t *arg)
{
 vga_poker_arg_t vga;
 vga.pixel_coors = *arg;

 if (ioctl(vga_poker_fd, VGA_POKER_SET_PIXELS, &vga)) {
   perror("ioctl(vga_poker_SET_BACKGROUND) failed");
   return;
 }
}

void set_tile(const vga_poker_tile_coords_t *arg)
{
 vga_poker_arg_t vga;
 vga.tile_coords = *arg;

 if (ioctl(vga_poker_fd, VGA_POKER_SET_TILE, &vga)) {
   perror("ioctl(vga_poker_SET_BACKGROUND) failed");
   return;
 }
}
```

```
/* resources found online to convert */
void load_palette(const char *fn) {
 FILE *f = fopen(fn, "r");
 if (!f) { perror("fopen palette"); return; }

 char line[32];
 for (int i = 0; i < 16; i++) {
     if (!fgets(line, sizeof line, f)) break;
     unsigned int word;
     if (sscanf(line, "%8x", &word) != 1) {
       printf("err\n");
     }
     vga_poker_palette_t p = {
         .index = i,
         .red   = (word >> 24) & 0xFF,
         .green = (word >> 16) & 0xFF,
         .blue  = (word >>  8) & 0xFF
     };
     set_palette(&p);
 }
 fclose(f);
}

/* resources found online to convert */
void load_tileset(const char *fn) {
 FILE *f = fopen(fn, "r");
 if (!f) { perror("fopen tileset"); return; }

 char line[128];
 int tile_id = 0, row = 0;
 vga_poker_pixels_t px = {0};

 while (fgets(line, sizeof line, f)) {

     // parse 8 hex values
     unsigned int vals[8];
     sscanf(line,
                 "%2x %2x %2x %2x %2x %2x %2x %2x",
                 &vals[0],&vals[1],&vals[2],&vals[3],
                 &vals[4],&vals[5],&vals[6],&vals[7]);

     // on first row of a new tile, set px.tile_id
     px.tile_id = tile_id;
     // copy this row into pixels[ row*8 + col ]
```

```
        for (int col = 0; col < 8; col++) {
            px.pixels[row*8 + col] = vals[col] & 0x0F;
        }

        row++;
        if (row == 8) {
            // we have one full tile → send to hardware
            set_pixels(&px);
            row = 0;
            tile_id++;
            if (tile_id >= 256) break;  // guard
        }
    }
    fclose(f);
}

/* resources found online to convert */
void load_tilemap(const char *fn) {
    FILE *f = fopen(fn, "r");
    if (!f) {
        perror("fopen tilemap");
        return;
    }

    char line[512];
    for (int row = 0; row < 60; row++) {
        if (!fgets(line, sizeof line, f)) {
            fprintf(stderr, "tilemap: missing row %d\n", row);
            break;
        }

        // Tokenize the line on whitespace
        char *tok = strtok(line, " \t\r\n");
        for (int col = 0; col < 80; col++) {

            // Parse two hex digits into an integer
            unsigned int tid = (unsigned int)strtoul(tok, NULL, 16);

            // Clamp to 0–255
            tid &= 0xFF;

            vga_poker_tile_coords_t tc = {
                .row     = row,
                .col     = col,
```

```
            .tile_id = tid
        };
        set_tile(&tc);
        tile_map[row][col] = tid;
        tok = strtok(NULL, " \t\r\n");
    }
 }

 fclose(f);
}

/* resources found online to convert */
int load_deck(const char *fn)
{
   FILE *f = fopen(fn, "r");
   if (!f) {
       fprintf(stderr, "file open error");
       return -1;
   }

   char line[256];
   for (int line_no = 0; line_no < 52 * CARD_TILE_ID_ROWS; line_no++) {
       if (!fgets(line, sizeof(line), f)) {
       fclose(f);
       return -1;
   }

   // Which card and which row within that card?
   int card    = line_no / CARD_TILE_ID_ROWS;
   int row_in  = line_no % CARD_TILE_ID_ROWS;

   // Tokenize on whitespace
   char *tok = strtok(line, " \t\r\n");
   for (int col = 0; col < CARD_TILE_ID_COLS; col++) {
       if (!tok) {
           fclose(f);
           return -1;
       }
       unsigned int tid = strtoul(tok, NULL, 16) & 0xFF;
       deck[card][row_in][col] = (uint8_t)tid;
       tok = strtok(NULL, " \t\r\n");
   }
   // any extra tokens on the line are simply ignored
   }
```

```
    fclose(f);
    return 0;
}

/* resources found online to convert */
int load_jokers(const char *fn)
{
    FILE *f = fopen(fn, "r");
    if (!f) {
        fprintf(stderr, "file open error");
        return -1;
    }

    char line[256];
    for (int line_no = 0; line_no < 10 * CARD_TILE_ID_ROWS; line_no++) {
        if (!fgets(line, sizeof(line), f)) {
        fclose(f);
        return -1;
    }

    // Which card and which row within that card?
    int card    = line_no / CARD_TILE_ID_ROWS;
    int row_in  = line_no % CARD_TILE_ID_ROWS;

    // Tokenize on whitespace
    char *tok = strtok(line, " \t\r\n");
    for (int col = 0; col < CARD_TILE_ID_COLS; col++) {
        if (!tok) {
            fclose(f);
            return -1;
        }
        unsigned int tid = strtoul(tok, NULL, 16) & 0xFF;
        jokers[card][row_in][col] = (uint8_t)tid;
        tok = strtok(NULL, " \t\r\n");
    }
    // any extra tokens on the line are simply ignored
    }

    fclose(f);
    return 0;
}

/* tile id 201 - 226 for A - Z */
```

```
uint8_t char_to_letter_id(char letter)
{
 if (letter == ' ') return 250;
 if (letter < 'A' || letter > 'Z') return -1;
 return letter + 136;
}

/* tile_id 228 - 237 for 0 - 9 */
uint8_t score_to_digit_id(char digit)
{
 if (digit < '0' || digit > '9') return -1;
 return digit + 179;
}

/* redraws the entire screen baesed on whats in the tilemap[][] */
void redraw_screen(void)
{
 for (int row = 0; row < 60; row++) {
   for (int col = 0; col < 80; col++) {
     vga_poker_tile_coords_t tile = {
       .row     = row,
       .col     = col,
       .tile_id = tile_map[row][col]
     };
     set_tile(&tile);
   }
 }
}

void draw_chip(char *digits)
{
 for (uint8_t col = 0; col < SCORE_DIGITS - 2; col++) {
   uint8_t tid = score_to_digit_id(digits[col]);
   tile_map[CHIP_MULT_CALC_ROW][CHIP_COL_START + col] = tid;
   vga_poker_tile_coords_t tile = {
     .row     = CHIP_MULT_CALC_ROW,
     .col     = CHIP_COL_START + col,
     .tile_id = tid
   };
   set_tile(&tile);
 }
}

void draw_mult(char *digits)
```

```
{
 for (uint8_t col = 0; col < SCORE_DIGITS - 2; col++) {
   uint8_t tid = score_to_digit_id(digits[col]);
   tile_map[CHIP_MULT_CALC_ROW][MULT_COL_START + col] = tid;
   vga_poker_tile_coords_t tile = {
     .row     = CHIP_MULT_CALC_ROW,
     .col     = MULT_COL_START + col,
     .tile_id = tid
   };
   set_tile(&tile);
 }
}
/* Draws the target score digits on screen given a array[6] of chars (i.e.
001256) */
void draw_target_score(char *digits)
{
 // MAP TILE_ID TO VAL
 for (uint8_t col = 0; col < SCORE_DIGITS; col++) {
   uint8_t tid = score_to_digit_id(digits[col]);
   tile_map[TARGET_SCORE_ROW][SCORE_COL_START + col] = tid;
   vga_poker_tile_coords_t tile = {
     .row     = TARGET_SCORE_ROW,
     .col     = SCORE_COL_START + col,
     .tile_id = tid
   };
   set_tile(&tile);
 }
}

/* Draws the round score digits on screen given a array[6] of chars (i.e.
001256) */
void draw_round_score(char *digits)
{
 for (uint8_t col = 0; col < SCORE_DIGITS; col++) {
   uint8_t tid = score_to_digit_id(digits[col]);
   tile_map[ROUND_SCORE_ROW][SCORE_COL_START + col] = tid;
   vga_poker_tile_coords_t tile = {
     .row     = ROUND_SCORE_ROW,
     .col     = SCORE_COL_START + col,
     .tile_id = tid
   };
   set_tile(&tile);
 }
}
```

```
/* Draws the hand type letters on screen given an array[15] where spaces are
included to center (i.e. ___FLUSH___) */
void draw_hand_type(char *hand_type)
{
 size_t len = strlen(hand_type);
 if (len > (HAND_TYPE_COL_END - HAND_TYPE_COL_START + 1))
     len = HAND_TYPE_COL_END - HAND_TYPE_COL_START + 1;

 for (uint8_t col = 0; col < len; col++) {
   uint8_t tid = char_to_letter_id(hand_type[col]);
   if (tid == 250) tid = 96;
   tile_map[HAND_TYPE_ROW][HAND_TYPE_COL_START + col] = tid;
   vga_poker_tile_coords_t tile = {
     .row     = HAND_TYPE_ROW,
     .col     = HAND_TYPE_COL_START + col,
     .tile_id = tid
   };
   set_tile(&tile);
 }
}

#define JOKER_NAME_ROW 49
#define JOKER_INFO_ROW_START 51
#define JOKER_INFO_ROW_END 55
#define JOKER_NAME_COL_START 2
#define JOKER_NAME_COL_END 15


/* Draws the hands_left digit on screen given a char digit */
void draw_hands_left(char digit)
{
 uint8_t tid = score_to_digit_id(digit);
 tile_map[HANDS_LEFT_ROW][HANDS_LEFT_COL] = tid;
 vga_poker_tile_coords_t tile = {
   .row     = HANDS_LEFT_ROW,
   .col     = HANDS_LEFT_COL,
   .tile_id = tid
 };
 set_tile(&tile);
}

/* Draws the discard digit on screen given a char digit */
void draw_discards(char digit)
```

```
{
 uint8_t tid = score_to_digit_id(digit);
 tile_map[DISCARD_ROW][DISCARD_COL] = tid;
 vga_poker_tile_coords_t tile = {
   .row     = DISCARD_ROW,
   .col     = DISCARD_COL,
   .tile_id = tid
 };
 set_tile(&tile);
}

/* Draws card in deck digits on screen given an array[2] where each index
contains a char digit */
void draw_cards_in_deck(char *cards_left)
{
 uint8_t tid_1 = score_to_digit_id(cards_left[0]);
 uint8_t tid_2 = score_to_digit_id(cards_left[1]);
 tile_map[CARDS_IN_DECK_ROW][CARDS_IN_DECK_COL] = tid_1;
 tile_map[CARDS_IN_DECK_ROW][CARDS_IN_DECK_COL + 1] = tid_2;
 vga_poker_tile_coords_t tile = {
   .row     = CARDS_IN_DECK_ROW,
   .col     = CARDS_IN_DECK_COL,
   .tile_id = tid_1
 };
 set_tile(&tile);
 tile.col = CARDS_IN_DECK_COL + 1;
 tile.tile_id = tid_2;
 set_tile(&tile);
}

/* Draws the corresponding round digit on screen given an round_number char */
void draw_round(char round_number)
{
 uint8_t tid = score_to_digit_id(round_number);
 tile_map[ANTE_ROUND_ROW][ROUND_COL] = tid;
 vga_poker_tile_coords_t tile = {
   .row     = ANTE_ROUND_ROW,
   .col     = ROUND_COL,
   .tile_id = tid
 };
 set_tile(&tile);
}

/* Draws the corresponding ante digit on screen given an ante_number char */
```

```
void draw_ante(char ante_number)
{
 // MAP TILE_ID TO VAL
 uint8_t tid = score_to_digit_id(ante_number);
 tile_map[ANTE_ROUND_ROW][ANTE_COL] = tid;
 vga_poker_tile_coords_t tile = {
   .row     = ANTE_ROUND_ROW,
   .col     = ANTE_COL,
   .tile_id = tid
 };
 set_tile(&tile);
}


/* cards_g is a 5 integer array containing all of deck indicies */
void draw_jokers(uint8_t *cards_g)
{
 uint8_t valid_jokers[5];
 for (uint8_t i = 0; i < 5; i++) {
   if (cards_g[i] == 10) valid_jokers[i] = 0;
   else valid_jokers[i] = 1;
 }
 for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
   for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
       tile_map[JOKER_ROW + row][JOKER_1_COL + col] = valid_jokers[0] == 1 ?
jokers[cards_g[0]][row][col] : card_slot[row][col];
       tile_map[JOKER_ROW + row][JOKER_2_COL + col] = valid_jokers[1] == 1 ?
jokers[cards_g[1]][row][col] : card_slot[row][col];
       tile_map[JOKER_ROW + row][JOKER_3_COL + col] = valid_jokers[2] == 1 ?
jokers[cards_g[2]][row][col] : card_slot[row][col];
       tile_map[JOKER_ROW + row][JOKER_4_COL + col] = valid_jokers[3] == 1 ?
jokers[cards_g[3]][row][col] : card_slot[row][col];
       tile_map[JOKER_ROW + row][JOKER_5_COL + col] = valid_jokers[4] == 1 ?
jokers[cards_g[4]][row][col] : card_slot[row][col];
   }
 }
 redraw_screen();
}
/* cards_g is a 5 integer array containing all of deck indicies */
void draw_played_cards(uint8_t *cards_g)
{
 /* optimization to only draw tiles you change
 vga_poker_tile_coords_t tile = {
       .row     = row,
```

```
        .col     = col,
        .tile_id = tile_map[row][col]
      };
      set_tile(&tile);
 */
uint8_t valid_cards[5];
for (uint8_t i = 0; i < 5; i++) {
  if (cards_g[i] == 52) valid_cards[i] = 0;
  else valid_cards[i] = 1;
}
 for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
   for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
     tile_map[PLAYED_ROW + row][PLAYED_1_COL + col] = valid_cards[0] == 1 ?
deck[cards_g[0]][row][col] : card_slot[row][col];
     tile_map[PLAYED_ROW + row][PLAYED_2_COL + col] = valid_cards[1] == 1 ?
deck[cards_g[1]][row][col] : card_slot[row][col];
     tile_map[PLAYED_ROW + row][PLAYED_3_COL + col] = valid_cards[2] == 1 ?
deck[cards_g[2]][row][col] : card_slot[row][col];
     tile_map[PLAYED_ROW + row][PLAYED_4_COL + col] = valid_cards[3] == 1 ?
deck[cards_g[3]][row][col] : card_slot[row][col];
     tile_map[PLAYED_ROW + row][PLAYED_5_COL + col] = valid_cards[4] == 1 ?
deck[cards_g[4]][row][col] : card_slot[row][col];
   }
 }
 redraw_screen();
}

/* Clears the played card positions one by one */
void clear_table(uint8_t amount)
{

 for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
   for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
       tile_map[PLAYED_ROW + row][PLAYED_1_COL + col] = card_slot[row][col];
       vga_poker_tile_coords_t tile = {
         .row     = PLAYED_ROW + row,
         .col     = PLAYED_1_COL + col,
         .tile_id = card_slot[row][col]
       };
       set_tile(&tile);
   }
 }
 sleep(1);
 if (amount >= 2) {
```

```
  for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
    for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
        tile_map[PLAYED_ROW + row][PLAYED_2_COL + col] = card_slot[row][col];
        vga_poker_tile_coords_t tile = {
          .row     = PLAYED_ROW + row,
          .col     = PLAYED_2_COL + col,
          .tile_id = card_slot[row][col]
        };
        set_tile(&tile);
    }
  }
  sleep(1);
}
if (amount >= 3) {
  for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
    for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
        tile_map[PLAYED_ROW + row][PLAYED_3_COL + col] = card_slot[row][col];
        vga_poker_tile_coords_t tile = {
          .row     = PLAYED_ROW + row,
          .col     = PLAYED_3_COL + col,
          .tile_id = card_slot[row][col]
        };
        set_tile(&tile);
    }
  }
  sleep(1);
}

if (amount >= 4) {
  for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
    for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
        tile_map[PLAYED_ROW + row][PLAYED_4_COL + col] = card_slot[row][col];
        vga_poker_tile_coords_t tile = {
          .row     = PLAYED_ROW + row,
          .col     = PLAYED_4_COL + col,
          .tile_id = card_slot[row][col]
        };
        set_tile(&tile);
    }
  }
  sleep(1);
}
if (amount >= 5) {
  for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
```

```
      for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
          tile_map[PLAYED_ROW + row][PLAYED_5_COL + col] = card_slot[row][col];
          vga_poker_tile_coords_t tile = {
            .row     = PLAYED_ROW + row,
            .col     = PLAYED_5_COL + col,
            .tile_id = card_slot[row][col]
          };
          set_tile(&tile);
      }
    }
    sleep(1);
  }

}

/* selected_arr: 8 integer array of either 0 or 1 where 1 denotes selected */
void draw_selected(uint8_t *selected_arr)
{
    /* green tile_id = 95, yellow_id =  91, blue_id = 94*/
    uint8_t tile_id = 94;
    for (uint8_t i = 0; i < HAND_8_COL_END - HAND_1_COL_START + 1; i++) {
      tile_map[SELECTED_ROW][SELECTED_COL + i] = tile_id + 1;
      vga_poker_tile_coords_t tile = {
        .row     = SELECTED_ROW,
        .col     = SELECTED_COL + i,
        .tile_id = tile_id + 1
      };
      set_tile(&tile);
    }
    for (uint8_t i = 0; i < 8; i++) {
      if (selected_arr[i] != 0) {
        uint8_t position = i * 5;
        tile_map[SELECTED_ROW][SELECTED_COL + 1 + position] = tile_id;
        vga_poker_tile_coords_t tile = {
          .row     = SELECTED_ROW,
          .col     = SELECTED_COL + 1 + position,
          .tile_id = tile_id
        };
        set_tile(&tile);

        tile.col = tile.col + 1;
        tile_map[SELECTED_ROW][SELECTED_COL + 2 + position] = tile_id;
        set_tile(&tile);
```

```
        tile.col = tile.col + 1;
        tile_map[SELECTED_ROW][SELECTED_COL + 3 + position] = tile_id;
        set_tile(&tile);

        if (i == 7) {
          tile.col = tile.col + 1;
          tile_map[SELECTED_ROW][SELECTED_COL + 4 + position] = tile_id;
          set_tile(&tile);

          tile.col = tile.col + 1;
          tile_map[SELECTED_ROW][SELECTED_COL + 5 + position] = tile_id;
          set_tile(&tile);
          tile.col = tile.col + 1;

          tile_map[SELECTED_ROW][SELECTED_COL + 6 + position] = tile_id;
          set_tile(&tile);

          tile.col = tile.col + 1;
          tile_map[SELECTED_ROW][SELECTED_COL + 7 + position] = tile_id;
          set_tile(&tile);

          tile.col = tile.col + 1;
          tile_map[SELECTED_ROW][SELECTED_COL + 8 + position] = tile_id;
          set_tile(&tile);
        }
      }

    }
}

/* Draws a yellow cursor underneath the card in hand given an index of what
card position you are at (i.e index 0 = left most card)*/
void draw_cursor(uint8_t index)
{
 /* green tile_id = 95, yellow_id =  91, red_id*/
 uint8_t position = index * 5;
 uint8_t tile_id = 91;
 uint8_t range = index == 7 ? 9 : 4;
 for (uint8_t i = 1; i < range; i++) {
   tile_map[CURSOR_ROW][CURSOR_COL + i + position] = tile_id;
   vga_poker_tile_coords_t tile = {
     .row     = CURSOR_ROW,
     .col     = CURSOR_COL + i + position,
     .tile_id = tile_id
```

```
    };
    set_tile(&tile);
  }
}
/* Draws a the backgournd underneath the card in hand given an index of what
card position you are at (i.e index 0 = left most card)*/
void clear_cursor(uint8_t index)
{
    /* green tile_id = 95, yellow_id =  91, red_id*/
    uint8_t position = index * 5;
    uint8_t tile_id = 95;
    uint8_t range = index == 7 ? 9 : 4;
    for (uint8_t i = 1; i < range; i++) {
      tile_map[CURSOR_ROW][CURSOR_COL + i + position] = tile_id;
      vga_poker_tile_coords_t tile = {
        .row     = CURSOR_ROW,
        .col     = CURSOR_COL + i + position,
        .tile_id = tile_id
      };
      set_tile(&tile);
    }
}

/* Draws the hand given a cards_g[8] where each index in cards_g is an index (0
- 51) to a corrseponding card.
* Example: if cards_g[0] = 0, then 0 would represent the 2 of clubs
*/
void draw_hand(uint8_t *cards_g)
{
 uint8_t valid_cards[8];
 for (uint8_t i = 0; i < 8; i++) {
   if (cards_g[i] == 52) valid_cards[i] = 0;
   else valid_cards[i] = 1;
 }
    for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
        for (uint8_t col = 0; col < CARD_TILE_ID_COLS / 2; col++) {
            tile_map[HAND_ROW_START + row][HAND_1_COL_START + col] =
valid_cards[0] == 1 ? deck[cards_g[0]][row][col] : card_slot[row][col];
            tile_map[HAND_ROW_START + row][HAND_2_COL_START + col] =
valid_cards[1] == 1 ? deck[cards_g[1]][row][col] : card_slot[row][col];
            tile_map[HAND_ROW_START + row][HAND_3_COL_START + col] =
valid_cards[2] == 1 ? deck[cards_g[2]][row][col] : card_slot[row][col];
            tile_map[HAND_ROW_START + row][HAND_4_COL_START + col] =
valid_cards[3] == 1 ? deck[cards_g[3]][row][col] : card_slot[row][col];
```

```
            tile_map[HAND_ROW_START + row][HAND_5_COL_START + col] =
valid_cards[4] == 1 ? deck[cards_g[4]][row][col] : card_slot[row][col];
            tile_map[HAND_ROW_START + row][HAND_6_COL_START + col] =
valid_cards[5] == 1 ? deck[cards_g[5]][row][col] : card_slot[row][col];
            tile_map[HAND_ROW_START + row][HAND_7_COL_START + col] =
valid_cards[6] == 1 ? deck[cards_g[6]][row][col] : card_slot[row][col];
        }
    }
    for (uint8_t row = 0; row < CARD_TILE_ID_ROWS; row++) {
      for (uint8_t col = 0; col < CARD_TILE_ID_COLS; col++) {
        tile_map[HAND_ROW_START + row][HAND_8_COL_START + col] = valid_cards[7]
== 1 ? deck[cards_g[7]][row][col] : card_slot[row][col];
      }
    }
    redraw_screen();
}
```

poker_logic.c

```
C/C++
#define _POSIX_C_SOURCE 200809L
#define _DEFAULT_SOURCE

#include "controller.h"
#include "poker.h"
#include "vga_poker.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <libusb-1.0/libusb.h>

#define RANK(c)  ((c) >> 2)
#define SUIT(c)  ((c) & 0x3)
#define MAKE_CARD(rank,suit)  (((rank) << 2) | (suit))
#define REPORT_LEN 8 // CONTROLLER BYTES REPORTED
#define DEBOUNCE_DELAY 150000 // 150ms debounce time in microseconds
```

```
#define COMMON 0
#define UNCOMMON 1
#define RARE 2
#define LEGENDARY 3

#define SMILEY_FACE 0   // Uncommon: +5 mult per face card
#define EVEN_STEVEN 1   // Uncommon: +4 mult per even rank card
#define ODD_TODD 2      // Uncommon: +31 chip per odd rank card
#define SUIT_BUNDLER 3  // Rare: Spades = Clubs, Diamonds = Hearts
#define THE_ONE 4       // Legendary: Next 7 hands are royal flushes (consumed)
#define BLUE_DOT 5      // Common: +50 chip
#define RED_DOT 6       // Common: +10 mult
#define GREEN_CHECK 7   // Rare: Need only 25% to pass round (consumed)
#define BLANK 8         // Common: Does nothing (funsies)
#define STEVIE_DOTT 9   // Rare: Even sum 40 mult, odd sum 2X chip

typedef uint8_t Card;

typedef struct {
   int chips;
   int multiplier;
} HandValue;

typedef struct {
   int small_blind;
   int big_blind;
   int boss;
} Blind;

typedef struct {
   int ante_number;
   Blind blinds;
} Ante;

typedef struct {
   float joker_chance;
   float rarity_probs[4];
} drop_probs;

Card deck_1[52];
Card drawed_cards[8]; // CARDS READY TO BE SELECTED
Card joker_cards[10]; // JOKER CARDS (ON OR OFF)
Card selected_cards[5]; // SELECTED CARDS (NOT PLAYED YET)
Card played_hand[5]; // PLAYED CARDS (PLAY OR DISCARD)
```

```
Ante antes[8]; // ANTES
uint8_t index_for_tiles_cards[8];

enum HandType {
   HIGH_CARD,
   PAIR,
   TWO_PAIR,
   THREE_OF_A_KIND,
   STRAIGHT,
   FLUSH,
   FULL_HOUSE,
   FOUR_OF_A_KIND,
   STRAIGHT_FLUSH,
   ROYAL_FLUSH
};

const int JOKER_RARITIES[10] = {
   UNCOMMON,  // 0: Smiley Face
   UNCOMMON,  // 1: Even Steven
   UNCOMMON,  // 2: Odd Todd
   RARE,      // 3: Suit Bundler
   LEGENDARY, // 4: The One
   COMMON,    // 5: Blue Dot
   COMMON,    // 6: Red Dot
   RARE,      // 7: Green Check
   COMMON,    // 8: Blank
   RARE       // 9: Stevie Dott
};

int game_play = 0;
int draw_index = 0;
int draw_amount = 8;
int hands_remaining = 4;
int discards_remaining = 4;
int game_round = 1;
int ante = 1;
int current_blind = 0;
int cursor = 0;
int num_selected_cards = 0;
unsigned char last_button = 0;
struct timespec last_button_time;

/* VGA VARIABLES */
char target_score_str[7];
```

```
char round_score_str[7];
char chip_str[5];
char mult_str[5];
char hands_left_str[2];
char discards_left_str[2];
char cards_in_deck_str[3];
char ante_str[2];
char round_str[2];
uint8_t cursor_position;
uint8_t clear_cursor_position;
uint8_t selected_array[8]; // Array of selected cards (0=not selected,
1=selected)

// Antes 1-3: Early game (lowest odds of rare/legendary)
const drop_probs EARLY_GAME = {0.50f, {0.40f, 0.35f, 0.15f, 0.10f}};
// Antes 4-6: Mid game (moderate odds of better jokers)
const drop_probs MID_GAME = {0.75f, {0.25f, 0.30f, 0.30f, 0.15f}};
// Antes 7-8: Late game (best odds of powerful jokers)
const drop_probs LATE_GAME = {1.00f, {0.15f, 0.20f, 0.40f, 0.25f}};


/* Hand Evaluation Functions */
HandValue get_hand_value(enum HandType hand_type);
int check_straight(int *rank_counts);
enum HandType evaluate_selected_cards();
enum HandType evaluate_hand(Card *hand, int num_cards);

/* Card and Deck Management Functions */
void init_deck();
void shuffle_deck();
Card draw_card();
int cards_remaining();
void draw_initial_pool_of_cards();
void draw_replacement_cards();
uint8_t* drawn_to_index();

/* Game State Management Functions */
void init_antes();
void init_jokers();
int play_selected_hand();
void discard_selected_cards();
int check_win_condition(int player_score);
void advance_game_state();
void round_reset();
```

```
void hard_reset();
void game_over();
void game_won();
void get_current_blind_info(char *blind_name, int *target_score);
void game_loop_vga();

/* VGA Display Functions */
const char* get_hand_name_vga(enum HandType hand);
char* get_target_score_vga(int target_score);
char* get_round_score_vga(int round_score);
char* get_chip_vga(int chip_value);
char* get_mult_vga(int multiplier);
char* get_hands_left_vga();
char* get_discards_left_vga();
char* get_cards_in_deck_vga();
char* get_ante_vga();
char* get_round_vga();
uint8_t get_cursor_position();
uint8_t get_clear_cursor_position();
void update_clear_cursor_position();
uint8_t* get_selected_cards_array();

/* Controller Input Functions */
void initialize_debounce();
int check_controller_input(unsigned char *report);
void toggle_card_selection();
void move_cursor(int direction);
void process_controller_input(unsigned char *report);

/* Score Calculation Functions */
int get_individual_card_chip(Card card);
int calculate_hand_type_specific_chips();

/* Joker Effect Functions */
int apply_smiley_face_joker(Card *hand, int num_cards, int base_multiplier);
int apply_even_steven_joker(Card *hand, int num_cards, int base_multiplier);
int apply_odd_todd_joker(Card *hand, int num_cards, int base_chips);
Card transform_card_suit_bundler(Card card);
void apply_suit_bundler_joker(Card *hand, int num_cards, Card
*transformed_hand);
void activate_the_one_joker();
int check_the_one_joker();
int apply_blue_dot_joker(int base_chips);
int apply_red_dot_joker(int base_multiplier);
```

```
int apply_green_check_joker(int target_score, int current_score, int
hands_left);
void apply_stevie_dott_joker(Card *hand, int num_cards, int *multiplier, int
*card_chips);
void apply_all_joker_effects(Card *hand, int num_cards, enum HandType
*hand_type, int *chips, int *multiplier, int target_score, int current_score,
int hands_left);

/* Joker Management Functions */
void joker_drop();
int count_active_jokers(Card joker_cards[10]);

/* Score Calculation Additional Functions */
int calculate_played_hand_chips(Card *hand, int num_cards, enum HandType
hand_type);

/* Main Function */
int main();

/*
* Given a HandType, Return The Hand Value
*/
HandValue get_hand_value(enum HandType hand_type)
{
   HandValue values[] = {
       {5, 1},     // HIGH_CARD
       {10, 2},    // PAIR
       {20, 2},    // TWO_PAIR
       {30, 3},    // THREE_OF_A_KIND
       {30, 4},    // STRAIGHT
       {35, 4},    // FLUSH
       {40, 4},    // FULL_HOUSE
       {60, 7},    // FOUR_OF_A_KIND
       {100, 10}, // STRAIGHT_FLUSH
       {200, 15}  // ROYAL_FLUSH
   };
   return values[hand_type];
}

/*
* Helper Function To Return String Version Of
* Played Hand For VGA
*/
const char* get_hand_name_vga(enum HandType hand)
```

```
{
   const char* names[] = {
       "   HIGH CARD   ",
       "     PAIR      ",
       "   TWO PAIR    ",
       "THREE OF A KIND",
       "   STRAIGHT    ",
       "     FLUSH     ",
       "   FULL HOUSE  ",
       " FOUR OF A KIND",
       " STRAIGHT FLUSH",
       "  ROYAL FLUSH  "
   };
   return names[hand];
}

/*
* Converts target score to 6-character string
* Pads with leading spaces
*/
char* get_target_score_vga(int target_score) {
   snprintf(target_score_str, sizeof(target_score_str), "%06d", target_score);
   return target_score_str;
}

/*
* Converts current round score to 6-character string
* Pads with leading spaces
*/
char* get_round_score_vga(int round_score) {
   snprintf(round_score_str, sizeof(round_score_str), "%06d", round_score);
   return round_score_str;
}

/*
* Converts chip value to 4-character string
* Pads with leading spaces
*/
char* get_chip_vga(int chip_value) {
   snprintf(chip_str, sizeof(chip_str), "%04d", chip_value);
   return chip_str;
}

/*
```

```
* Converts multiplier to 4-character string
* Pads with leading spaces
*/
char* get_mult_vga(int multiplier) {
   snprintf(mult_str, sizeof(mult_str), "%04d", multiplier);
   return mult_str;
}

/*
* Converts hands left to 1-character string
*/
char* get_hands_left_vga() {
   snprintf(hands_left_str, sizeof(hands_left_str), "%d", hands_remaining);
   return hands_left_str;
}

/*
* Converts discards left to 1-character string
*/
char* get_discards_left_vga() {
   snprintf(discards_left_str, sizeof(discards_left_str), "%d",
discards_remaining);
   return discards_left_str;
}

/*
* Converts cards in deck to 2-character string
*/
char* get_cards_in_deck_vga() {
   int cards = cards_remaining();
   if (cards < 10) {
       snprintf(cards_in_deck_str, sizeof(cards_in_deck_str), "0%d", cards);
   } else {
       snprintf(cards_in_deck_str, sizeof(cards_in_deck_str), "%d", cards);
   }

   return cards_in_deck_str;
}

/*
* Converts ante to 1-character string
*/
char* get_ante_vga() {
   snprintf(ante_str, sizeof(ante_str), "%d", ante);
```

```
    return ante_str;
}

/*
* Converts round to 1-character string
*/
char* get_round_vga() {
    snprintf(round_str, sizeof(round_str), "%d", game_round);
    return round_str;
}

/*
* Gets current cursor position (0-7)
*/
uint8_t get_cursor_position() {
    return (uint8_t)cursor;
}

/*
* Gets previous cursor position for clearing
* Should be called before updating cursor
*/
uint8_t get_clear_cursor_position() {
    return clear_cursor_position;
}

/*
* Updates the clear cursor position to current cursor
* Call this whenever cursor is about to change
*/
void update_clear_cursor_position() {
    clear_cursor_position = cursor;
}

/*
* Builds an array of selected card indicators
* Returns an array of 8 uint8_t values:
* 0 = not selected, 1 = selected
*/
uint8_t* get_selected_cards_array() {
    // Initialize all to not selected
    for (int i = 0; i < 8; i++) {
        selected_array[i] = 0;
    }
```

```
    // Mark selected cards
    for (int i = 0; i < num_selected_cards; i++) {
        for (int j = 0; j < draw_amount; j++) {
            if (drawed_cards[j] == selected_cards[i]) {
                selected_array[j] = 1;
                break;
            }
        }
    }

    return selected_array;
}


/*
* Initialize Deck of Cards
* Suit[0, 1, 2, 3] = [Clove, Diamond, Spade, Heart]
* Rank[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] = [2, 3, 4, 5, 6, 7, 8, 9, 10,
J, Q, K, A]
*/
void init_deck()
{
    int idx = 0;
    for (int suit = 0; suit < 4; suit++) {
        for (int rank = 0; rank < 13; rank++) {
            deck_l[idx] = MAKE_CARD(rank, suit);
            idx++;
        }
    }
}

/*
* Initialize The Antes And Blinds
* Small Blinds = Base Values
* Big Blinds = Base Values * 1.5
* Boss Blinds = Base Values * 2
*/
void init_antes()
{
    int base_values[] = {300, 800, 2000, 5000, 11000, 20000, 35000, 50000};

    for (int i = 0; i < 8; i++) {
        antes[i].ante_number = i + 1;
```

```
        antes[i].blinds.small_blind = base_values[i];
        antes[i].blinds.big_blind = (int)(base_values[i] * 1.5);
        antes[i].blinds.boss = base_values[i] * 2;
    }
}

/*
* Initialize Joker Cards
* Initialized to 0.
*/
void init_jokers()
{
    for (int i = 0; i < 10; i++) {
        joker_cards[i] = 0;
    }
}

/*
* Shuffles the Deck of Cards
* Uses Fisher-Yates (Knuth) Shuffle Algorithm
*/
void shuffle_deck()
{
    /* NEED THIS FOR RANDOM SEEDING */
    srand(time(NULL));
    for (int i = 51; i > 0; i--) {
        // Generate random index between 0 and i (inclusive)
        int j = rand() % (i + 1);
        // Swap deck[i] with deck[j]
        Card temp = deck_l[i];
        deck_l[i] = deck_l[j];
        deck_l[j] = temp;
    }
}

/*
* Draws A Card From The Deck
*
* On Success: Returns The Topmost Card
* On Error: Returns 0xFF
*/
Card draw_card()
{
    if (draw_index < 52 && cards_remaining() > 0) {
```

```
        Card card = deck_l[draw_index];
        draw_index++;
        return card;
    }
    return 0xFF;
}

/*
* Checks The Amount Of Cards Remaining In The Deck
*
* Returns The Amount Of Cards Remaining In The Deck
*/
int cards_remaining()
{
    return 52 - draw_index;
}

/*
* Draw the initial pool of cards. Initially 8 (draw_amount).
* Can be expanded to 10.
*/
void draw_initial_pool_of_cards()
{
    for (int i = 0; i < draw_amount; i++) {
        drawed_cards[i] = draw_card();
    }
}

/*
* Takes An Array Of The Selected Card Ranks
* Sees If The Selected Cards Form a Straight
*
* Returns 1 If A Straight Is Formed
* Returns 0 If No Straight If Formed
*/
int check_straight(int *rank_counts)
{
    // First, count how many different ranks we have
    int unique_ranks = 0;
    for (int i = 0; i < 13; i++) {
        if (rank_counts[i] > 0) {
            unique_ranks++;
        }
    }
```

```
    // A straight must have exactly 5 different ranks
    if (unique_ranks != 5) {
        return 0;
    }

    // Check for regular straight (5 consecutive ranks)
    int consecutive = 0;
    for (int i = 0; i < 13; i++) {
        if (rank_counts[i] > 0) {
            consecutive++;
            if (consecutive == 5) {
                return 1;
            }
        } else {
            consecutive = 0;  // Reset counter when we find a gap
        }
    }

    // Check for Ace-low straight (A-2-3-4-5)
    // Must have exactly these 5 ranks and no others
    if (rank_counts[12] > 0 && rank_counts[0] > 0 && rank_counts[1] > 0 &&
        rank_counts[2] > 0 && rank_counts[3] > 0) {
        // Verify we don't have any other ranks
        for (int i = 4; i < 12; i++) {
            if (rank_counts[i] > 0) {
                return 0;
            }
        }
        return 1;
    }

    return 0;
}

/*
* Evaluate Selected Cards
* Returns The Hand Type (enum HandType) for any number of cards (1-5)
*/
enum HandType evaluate_selected_cards()
{
    // Return HIGH_CARD if no cards selected
    if (num_selected_cards == 0) {
        return HIGH_CARD;
```

```
    }

    // Count ranks and suits
    int rank_counts[13] = {0};  // Index = rank (0-12)
    int suit_counts[4] = {0};   // Index = suit (0-3)

    for (int i = 0; i < num_selected_cards; i++) {
        rank_counts[RANK(selected_cards[i])]++;
        suit_counts[SUIT(selected_cards[i])]++;
    }

    // For flush, we need 5 cards
    int is_flush = 0;
    if (num_selected_cards == 5) {
        for (int i = 0; i < 4; i++) {
            if (suit_counts[i] == 5) {
                is_flush = 1;
                break;
            }
        }
    }

    // For straight, we need 5 cards
    int is_straight = (num_selected_cards == 5) ? check_straight(rank_counts) :
0;

    // Count pairs, trips, quads
    int pairs = 0, trips = 0, quads = 0;

    for (int i = 0; i < 13; i++) {
        if (rank_counts[i] == 2) pairs++;
        if (rank_counts[i] == 3) trips++;
        if (rank_counts[i] == 4) quads++;
    }

    // Royal flush and straight flush require 5 cards
    if (num_selected_cards == 5 && is_flush && is_straight) {
        // Check if it's a royal (10-J-Q-K-A)
        if (rank_counts[8] > 0 && rank_counts[9] > 0 && rank_counts[10] > 0 &&
            rank_counts[11] > 0 && rank_counts[12] > 0) {
            return ROYAL_FLUSH;
        }
        return STRAIGHT_FLUSH;
    }
```

```
    // Evaluate best possible hand based on cards played
    if (quads > 0) return FOUR_OF_A_KIND;
    if (trips > 0 && pairs > 0) return FULL_HOUSE;
    if (is_flush) return FLUSH;
    if (is_straight) return STRAIGHT;
    if (trips > 0) return THREE_OF_A_KIND;
    if (pairs == 2) return TWO_PAIR;
    if (pairs == 1) return PAIR;

    return HIGH_CARD;
}

/*
* Given a Hand And Num Cards
* Evaluate The Type Of Hand
*/
enum HandType evaluate_hand(Card *hand, int num_cards)
{
    // Handle empty hand or more than 5 cards
    if (num_cards <= 0 || num_cards > 5) {
        return HIGH_CARD;
    }

    // Count ranks and suits
    int rank_counts[13] = {0};  // Index = rank (0-12)
    int suit_counts[4] = {0};   // Index = suit (0-3)

    for (int i = 0; i < num_cards; i++) {
        if (hand[i] == 0) continue; // Skip empty slots

        rank_counts[RANK(hand[i])]++;
        suit_counts[SUIT(hand[i])]++;
    }

    // Check for flush - requires exactly 5 cards of same suit
    int is_flush = 0;
    if (num_cards == 5) {  // Flush requires exactly 5 cards
        for (int i = 0; i < 4; i++) {
            if (suit_counts[i] == 5) {
                is_flush = 1;
                break;
            }
        }
```

```
    }

    // For straight, we need 5 cards
    int is_straight = (num_cards == 5) ? check_straight(rank_counts) : 0;

    // Count pairs, trips, quads
    int pairs = 0, trips = 0, quads = 0;

    for (int i = 0; i < 13; i++) {
        if (rank_counts[i] == 2) pairs++;
        if (rank_counts[i] == 3) trips++;
        if (rank_counts[i] == 4) quads++;
    }

    // Royal flush and straight flush require 5 cards
    if (num_cards == 5 && is_flush && is_straight) {
        // Check if it's a royal (10-J-Q-K-A)
        if (rank_counts[8] > 0 && rank_counts[9] > 0 && rank_counts[10] > 0 &&
            rank_counts[11] > 0 && rank_counts[12] > 0) {
            return ROYAL_FLUSH;
        }
        return STRAIGHT_FLUSH;
    }

    // Evaluate best possible hand based on cards played
    if (quads > 0) return FOUR_OF_A_KIND;
    if (trips > 0 && pairs > 0) return FULL_HOUSE;
    if (is_flush) return FLUSH;  // Only possible with 5 cards
    if (is_straight) return STRAIGHT;  // Only possible with 5 cards
    if (trips > 0) return THREE_OF_A_KIND;
    if (pairs == 2) return TWO_PAIR;
    if (pairs == 1) return PAIR;

    return HIGH_CARD;
}

/*
* Get Individual Card Chip Value
* 2-9: Face value (2 chips, 3 chips, etc.)
* 10, J, Q, K: 10 chips each
* A: 11 chips
*/
int get_individual_card_chip(Card card)
{
```

```
    int rank = RANK(card);

    if (rank >= 0 && rank <= 7) {        // 2-9 (rank 0-7)
        return rank + 2;                 // rank 0 = 2, rank 1 = 3, etc.
    } else if (rank >= 8 && rank <= 11) { // 10, J, Q, K (rank 8-11)
        return 10;
    } else if (rank == 12) {             // A (rank 12)
        return 11;
    }
    return 0;
}

/*
* Calculate Chip Values for Cards that Make Up the Hand Type
* Works with any number of cards (1-5)
*/
int calculate_hand_type_specific_chips()
{
    // Handle case of no cards
    if (num_selected_cards == 0) {
        return 0;
    }

    // Count ranks
    int rank_counts[13] = {0};
    Card cards_by_rank[13][4]; // Store which cards have each rank
    int count_by_rank[13] = {0};

    for (int i = 0; i < num_selected_cards; i++) {
        int rank = RANK(selected_cards[i]);
        rank_counts[rank]++;
        cards_by_rank[rank][count_by_rank[rank]] = selected_cards[i];
        count_by_rank[rank]++;
    }

    enum HandType hand_type = evaluate_selected_cards();
    int total_chips = 0;

    switch(hand_type) {
        case ROYAL_FLUSH:
        case STRAIGHT_FLUSH:
        case FULL_HOUSE:
        case FLUSH:
        case STRAIGHT:
```

```
            // All cards make up these hands
            for (int i = 0; i < num_selected_cards; i++) {
                total_chips += get_individual_card_chip(selected_cards[i]);
            }
            break;

        case FOUR_OF_A_KIND:
            // Only the matching cards count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 4) {
                    for (int j = 0; j < 4; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                } else if (rank_counts[i] == 3) {
                    // If we only have 3 cards that match
                    for (int j = 0; j < 3; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                } else if (rank_counts[i] == 2) {
                    // If we only have 2 cards that match
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                }
            }
            break;

        case THREE_OF_A_KIND:
            // Only the matching cards count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 3) {
                    for (int j = 0; j < 3; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                } else if (rank_counts[i] == 2) {
                    // If we only have 2 cards of a kind
```

```
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                }
            }
            break;

        case TWO_PAIR:
            // Only the cards that form the pairs count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 2) {
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                }
            }
            break;

        case PAIR:
            // Only the matching cards count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 2) {
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                }
            }
            break;

        case HIGH_CARD: {
            // Only the highest card counts
            int highest_rank = -1;
            Card highest_card = 0;
            for (int i = 0; i < num_selected_cards; i++) {
                int rank = RANK(selected_cards[i]);
                if (rank > highest_rank) {
                    highest_rank = rank;
                    highest_card = selected_cards[i];
                }
```

```
            }
            total_chips = get_individual_card_chip(highest_card);
            break;
        }
    }

    return total_chips;
}

/*
* Play Selected Cards
*/
int play_selected_hand()
{
    // Copy selected cards to played_hand
    for (int i = 0; i < num_selected_cards; i++) {
        played_hand[i] = selected_cards[i];
    }

    // Clear remaining slots in played_hand
    for (int i = num_selected_cards; i < 5; i++) {
        played_hand[i] = 0;
    }

    // Evaluate hand and calculate score
    enum HandType hand_type = evaluate_selected_cards();
    HandValue hand_value = get_hand_value(hand_type);
    int hand_specific_chips = calculate_hand_type_specific_chips();
    int total_chip_value = hand_value.chips + hand_specific_chips;
    int score = total_chip_value * hand_value.multiplier;

    // Mark played cards as invalid in drawed_cards
    for (int i = 0; i < num_selected_cards; i++) {
        for (int j = 0; j < draw_amount; j++) {
            if (drawed_cards[j] == selected_cards[i]) {
                drawed_cards[j] = 0xFF; // Mark as played
                break;
            }
        }
    }

    // Clear selected cards
    for (int i = 0; i < num_selected_cards; i++) {
        selected_cards[i] = 0;
```

```
    }

    num_selected_cards = 0; // Reset selection count
    hands_remaining--;

    // Reset cursor to a valid position
    cursor = 0;

    return score;
}

/*
* Discard Selected Cards
* Fixed to properly discard any number of cards
*/
void discard_selected_cards()
{
    // Copy selected cards to played_hand for tracking
    for (int i = 0; i < num_selected_cards; i++) {
        played_hand[i] = selected_cards[i];
    }

    // Clear remaining slots in played_hand
    for (int i = num_selected_cards; i < 5; i++) {
        played_hand[i] = 0;
    }

    // Mark selected cards as invalid in drawed_cards
    for (int i = 0; i < num_selected_cards; i++) {
        for (int j = 0; j < draw_amount; j++) {
            if (drawed_cards[j] == played_hand[i]) {
                drawed_cards[j] = 0xFF; // Mark as discarded
                break;
            }
        }
    }

    // Clear selected cards
    for (int i = 0; i < num_selected_cards; i++) {
        selected_cards[i] = 0;
    }

    num_selected_cards = 0; // Reset selection count
    discards_remaining--;
```

```
}

/*
* Draws Replacement Cards After Play/Dicarded
*/
void draw_replacement_cards()
{
   for (int i = 0; i < draw_amount; i++) {
       if (drawed_cards[i] == 0xFF) {
           Card new_card = draw_card();
           if (new_card != 0xFF) {  // Valid card drawn
               drawed_cards[i] = new_card;
           }
       }
   }
}

/*
* Function That Checks If The Player Has Reached The Target
* Score For The Ante/Blind/Round They Are On
*/
int check_win_condition(int player_score)
{
   Ante current_ante = antes[ante - 1];
   int target_score;

   if (current_blind == 0) {
       target_score = current_ante.blinds.small_blind;
   } else if (current_blind == 1) {
       target_score = current_ante.blinds.big_blind;
   } else {
       target_score = current_ante.blinds.boss;
   }

   return player_score >= target_score;
}

/*
* Advance Game State Function
*
* Called After User Beats A Blind
*/
void advance_game_state()
{
```

```
    current_blind++;
    game_round++;
    if (game_round > 3) {
        game_round = 1;
    }

    if (current_blind > 2) {
        current_blind = 0;
        ante++;

        if (ante > 8) {
            game_won();
        }
    }
    round_reset();
}

/*
* Gets The Current Blind Name and Target Score
*
* Ie. Small Blind, Big Blind, Boss Blind
* Retrieves The Target Score Needed To Beat Blind
*/
void get_current_blind_info(char *blind_name, int *target_score)
{
    Ante current_ante = antes[ante - 1];

    if (current_blind == 0) {
        strcpy(blind_name, "Small Blind");
        *target_score = current_ante.blinds.small_blind;
    } else if (current_blind == 1) {
        strcpy(blind_name, "Big Blind");
        *target_score = current_ante.blinds.big_blind;
    } else {
        strcpy(blind_name, "Boss");
        *target_score = current_ante.blinds.boss;
    }
}

/*
* Initialize the debounce timer
*/
void initialize_debounce() {
    clock_gettime(CLOCK_MONOTONIC, &last_button_time);
```

```
}

/*
* Check If Controller Report Matches Expected Pattern
*/
int check_controller_input(unsigned char *report)
{
   unsigned char patterns[][8] = {
       {0x01, 0x7f, 0x7f, 0x7f, 0x7f, 0x0f, 0x00, 0x00}, // NONE
       {0x01, 0x7f, 0x7f, 0x00, 0x7f, 0x0f, 0x00, 0x00}, // LEFT
       {0x01, 0x7f, 0x7f, 0xff, 0x7f, 0x0f, 0x00, 0x00}, // RIGHT
       {0x01, 0x7f, 0x7f, 0x7f, 0x00, 0x0f, 0x00, 0x00}, // UP
       {0x01, 0x7f, 0x7f, 0x7f, 0xff, 0x0f, 0x00, 0x00}, // DOWN
       {0x01, 0x7f, 0x7f, 0x7f, 0x7f, 0x2f, 0x00, 0x00}, // A
       {0x01, 0x7f, 0x7f, 0x7f, 0x7f, 0x4f, 0x00, 0x00}, // B
       {0x01, 0x7f, 0x7f, 0x7f, 0x7f, 0x0f, 0x10, 0x00}, // SELECT
       {0x01, 0x7f, 0x7f, 0x7f, 0x7f, 0x0f, 0x20, 0x00}  // START
   };

   enum {NONE = 0, LEFT, RIGHT, UP, DOWN, A, B, SELECT, START};

   for (int i = 0; i < 9; i++) {
       int match = 1;
       for (int j = 0; j < 8; j++) {
           if (report[j] != patterns[i][j]) {
               match = 0;
               break;
           }
       }
       if (match) return i;
   }

   return NONE;
}

/*
* Toggle Card Selection Based on Cursor Position
*/
void toggle_card_selection()
{
   // Check if card is already selected
   int is_selected = 0;
   int selected_index = -1;
```

```
    for (int i = 0; i < num_selected_cards; i++) {
        if (selected_cards[i] == drawed_cards[cursor]) {
            is_selected = 1;
            selected_index = i;
            break;
        }
    }

    if (is_selected) {
        // Remove from selection
        for (int i = selected_index; i < num_selected_cards - 1; i++) {
            selected_cards[i] = selected_cards[i + 1];
        }
        num_selected_cards--;
        selected_cards[num_selected_cards] = 0; // Clear the slot
    } else {
        // Add to selection (if not full)
        if (num_selected_cards < 5) {
            selected_cards[num_selected_cards] = drawed_cards[cursor];
            num_selected_cards++;
        }
    }
}

/*
* Move Cursor Left or Right
*/
void move_cursor(int direction)
{
    int original_cursor = cursor;
    int valid_position_found = 0;
    int attempts = 0;

    // Try to find a valid position up to draw_amount attempts
    while (!valid_position_found && attempts < draw_amount) {
        if (direction > 0) { // Move right
            cursor = (cursor + 1) % draw_amount;
        } else { // Move left
            cursor = (cursor - 1 + draw_amount) % draw_amount;
        }

        // Check if this position has a valid card
        if (drawed_cards[cursor] != 0xFF) {
            valid_position_found = 1;
```

```
        }

        attempts++;
    }

    // If no valid position found, reset to original position (should never
happen if at least one card exists)
    if (!valid_position_found) {
        cursor = original_cursor;
    }
}


/*
* Process Controller Input
* Updated to handle debouncing and support playing 1-5 cards
*/
void process_controller_input(unsigned char *report)
{
    int button = check_controller_input(report);

    if (button == 0) {
        last_button = 0;
        return;
    } else if ((button == last_button)) {
        return;
    }

    clock_gettime(CLOCK_MONOTONIC, &last_button_time);
    last_button = button;
    if (!game_play) {
        if (button == 8) {
            load_tilemap("tilemap.hex");
            game_play = 1;
        }
    } else if (game_play) {

        switch(button) {
            case 2: // RIGHT
                move_cursor(1);
                break;

            case 1: // LEFT
                move_cursor(-1);
```

```
                break;

            case 5: // A (Select/Deselect)
                toggle_card_selection();
                break;

            case 6: // B (Remove from selection)
                // Remove the card at cursor from selection if it's selected
                for (int i = 0; i < num_selected_cards; i++) {
                    if (selected_cards[i] == drawed_cards[cursor]) {
                        for (int j = i; j < num_selected_cards - 1; j++) {
                            selected_cards[j] = selected_cards[j + 1];
                        }
                        num_selected_cards--;
                        selected_cards[num_selected_cards] = 0;
                        break;
                    }
                }
                break;

            case 8: // START (Play selected cards - 1 to 5 cards allowed)
                if (game_play && num_selected_cards > 0 && num_selected_cards <=
5) {
                    play_selected_hand();
                }
                break;

            case 7: // SELECT (Discard selected cards)
                if (num_selected_cards > 0 && discards_remaining > 0) {
                    discard_selected_cards();
                    draw_replacement_cards();
                }
                break;
        }
    }
}

/*
* Map drawn cards to tile indices for display
*/
uint8_t *drawn_to_index()
{
    for (int i = 0; i < draw_amount; i++) {
        if (SUIT(drawed_cards[i]) == 0) {
```

```
            index_for_tiles_cards[i] = RANK(drawed_cards[i]);
        } else if (SUIT(drawed_cards[i]) == 1) {
            index_for_tiles_cards[i] = (13) + RANK(drawed_cards[i]);
        } else if (SUIT(drawed_cards[i]) == 2) {
            index_for_tiles_cards[i] = (13 * 2) + RANK(drawed_cards[i]);
        } else if (SUIT(drawed_cards[i]) == 3) {
            index_for_tiles_cards[i] = (13 * 3) + RANK(drawed_cards[i]);
        }
    }
    return index_for_tiles_cards;
}

/*
* Reset Function
*
* Called When User Wins a Game Round
*/
void round_reset()
{
    draw_index = 0;
    draw_amount = 8;
    hands_remaining = 4;
    discards_remaining = 4;
    cursor = 0;
    init_deck();
    init_antes();
    shuffle_deck();
    draw_initial_pool_of_cards();
    drawn_to_index();
}

/*
* HARD Reset Function
*
* Called When Restarting Game
*/
void hard_reset()
{
    ante = 1;
    current_blind = 0;
    game_round = 1;
    draw_index = 0;
    draw_amount = 8;
    hands_remaining = 4;
```

```
    discards_remaining = 4;
    init_deck();
    init_antes();
    init_jokers();
    shuffle_deck();
    draw_initial_pool_of_cards();
    drawn_to_index();
}

/*
* Game Over Function
* Called When User Loses
*/
void game_over()
{
    load_tilemap("256-Game-Over.hex");
    sleep(3);
    load_tilemap("256-title.hex");
    game_play = 0;
}

/*
* Game Won Function.
*
* Called After User Beats Ante 8
*/
void game_won()
{
    // DRAW SOMETHING
    load_tilemap("256-title.hex");
    game_play = 0;
}

/*
* Calculate Chip Values for Cards that Make Up a Played Hand
* Works with any number of cards (1-5)
* Takes the hand array, number of cards, and hand type as parameters
*/
int calculate_played_hand_chips(Card *hand, int num_cards, enum HandType
hand_type)
{
    if (num_cards == 0) {
        return 0;
    }
```

```
    // Count ranks
    int rank_counts[13] = {0};
    Card cards_by_rank[13][4]; // Store which cards have each rank
    int count_by_rank[13] = {0};

    for (int i = 0; i < num_cards; i++) {
        if (hand[i] == 0) continue; // Skip empty slots

        int rank = RANK(hand[i]);
        rank_counts[rank]++;
        cards_by_rank[rank][count_by_rank[rank]] = hand[i];
        count_by_rank[rank]++;
    }

    int total_chips = 0;

    switch(hand_type) {
        case ROYAL_FLUSH:
        case STRAIGHT_FLUSH:
        case FULL_HOUSE:
        case FLUSH:
        case STRAIGHT:
            // All cards make up these hands
            for (int i = 0; i < num_cards; i++) {
                if (hand[i] != 0) { // Skip empty slots
                    total_chips += get_individual_card_chip(hand[i]);
                }
            }
            break;

        case FOUR_OF_A_KIND:
            // Only the matching cards count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 4) {
                    for (int j = 0; j < 4; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                } else if (rank_counts[i] == 3) {
                    // If we only have 3 cards that match
                    for (int j = 0; j < 3; j++) {
```

```
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                } else if (rank_counts[i] == 2) {
                    // If we only have 2 cards that match
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                }
            }
            break;

        case THREE_OF_A_KIND:
            // Only the matching cards count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 3) {
                    for (int j = 0; j < 3; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                } else if (rank_counts[i] == 2) {
                    // If we only have 2 cards of a kind
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                }
            }
            break;

        case TWO_PAIR:
            // Only the cards that form the pairs count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 2) {
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                }
```

```
            }
            break;

        case PAIR:
            // Only the matching cards count
            for (int i = 0; i < 13; i++) {
                if (rank_counts[i] == 2) {
                    for (int j = 0; j < 2; j++) {
                        total_chips +=
get_individual_card_chip(cards_by_rank[i][j]);
                    }
                    break;
                }
            }
            break;

        case HIGH_CARD: {
            // Only the highest card counts
            int highest_rank = -1;
            Card highest_card = 0;
            for (int i = 0; i < num_cards; i++) {
                if (hand[i] == 0) continue; // Skip empty slots

                int rank = RANK(hand[i]);
                if (rank > highest_rank) {
                    highest_rank = rank;
                    highest_card = hand[i];
                }
            }
            total_chips = get_individual_card_chip(highest_card);
            break;
        }
    }

    return total_chips;
}

/*
* Counts how many active jokers the player currently has
* Returns a count of all joker_cards[] elements with value 1
*/
int count_active_jokers(Card joker_cards[10])
{
    int count = 0;
```

```
    for (int i = 0; i < 10; i++) {
        if (joker_cards[i] == 1) {
            count++;
        }
    }
    return count;
}

/*
* Joker Probability Drawer
* Determines if a joker drops after beating a blind/round
* Selects which joker to give based on rarity tables
*/
void joker_drop()
{
    // Get current ante to determine game stage
    int current_ante = ante;

    // Check if player already has maximum jokers
    if (count_active_jokers(joker_cards) >= 5) {
        return; // No more jokers can be added
    }

    // Select probability table based on game stage
    drop_probs probs;
    if (current_ante <= 3) {
        probs = EARLY_GAME;
    } else if (current_ante <= 6) {
        probs = MID_GAME;
    } else {
        probs = LATE_GAME;
    }

    // Determine if a joker drops at all
    float roll = (float)rand() / RAND_MAX;
    if (roll > probs.joker_chance) {
        return; // No joker drops
    }

    // Determine the rarity of the joker
    roll = (float)rand() / RAND_MAX;
    int rarity;
    float cumulative = 0.0f;
    for (rarity = 0; rarity < 4; rarity++) {
```

```
        cumulative += probs.rarity_probs[rarity];
        if (roll <= cumulative) break;
    }

    // Find available jokers of the chosen rarity
    int available_jokers[10];
    int count = 0;

    for (int i = 0; i < 10; i++) {
        if (JOKER_RARITIES[i] == rarity && joker_cards[i] == 0) {
            available_jokers[count++] = i;
        }
    }

    // If no jokers of chosen rarity are available, try any rarity
    if (count == 0) {
        for (int i = 0; i < 10; i++) {
            if (joker_cards[i] == 0) {
                available_jokers[count++] = i;
            }
        }

        // If all jokers are already active, just return
        if (count == 0) {
            return;
        }
    }

    // Select a random joker from the available pool
    int selected_joker = available_jokers[rand() % count];

    // Give the joker to the player
    joker_cards[selected_joker] = 1;
}

/*
* Smiley Face (Uncommon) ~ Plus 5 Mult Per Face Card Played
* Adds 5 to multiplier for each face card (J, Q, K) in the played hand
*/
int apply_smiley_face_joker(Card *hand, int num_cards, int base_multiplier)
{
    if (joker_cards[0] == 0) { // Check if this joker is not active
        return base_multiplier;
    }
```

```
    int additional_mult = 0;

    for (int i = 0; i < num_cards; i++) {
        int rank = RANK(hand[i]);
        // J=9, Q=10, K=11 (ranks 0-12)
        if (rank >= 9 && rank <= 11) {
            additional_mult += 5;
        }
    }

    return base_multiplier + additional_mult;
}

/*
* Even Steven (Uncommon) ~ Plus 4 Mult Per Even Rank Card Played
* Adds 4 to multiplier for each even-ranked card (2,4,6,8,10,Q) in the played
hand
*/
int apply_even_steven_joker(Card *hand, int num_cards, int base_multiplier)
{
    if (joker_cards[1] == 0) { // Check if this joker is not active
        return base_multiplier;
    }

    int additional_mult = 0;

    for (int i = 0; i < num_cards; i++) {
        int rank = RANK(hand[i]);
        // Even ranks: 0(2), 2(4), 4(6), 6(8), 8(10)
        if (rank % 2 == 0 && rank < 9) {
            additional_mult += 4;
        }
    }

    return base_multiplier + additional_mult;
}

/*
* Odd Todd (Uncommon) ~ Plus 31 Chip per Odd Rank Card Played
* Adds 31 chips for each odd-ranked card (A,3,5,7,9) in the played hand
*/
int apply_odd_todd_joker(Card *hand, int num_cards, int base_chips)
{
```

```
    if (joker_cards[2] == 0) { // Check if this joker is not active
        return base_chips;
    }

    int additional_chips = 0;

    for (int i = 0; i < num_cards; i++) {
        int rank = RANK(hand[i]);
        // Odd ranks: 1(3), 3(5), 5(7), 7(9), 12(A is odd in value)
        if ((rank % 2 == 1 || rank == 12) && rank != 9 && rank != 11) {
            additional_chips += 31;
        }
    }

    return base_chips + additional_chips;
}

/*
* Suit Bundler (Rare) - Spade Equals Clubs And Diamonds Equal Hearts
* Makes spades equal to clubs and diamonds equal to hearts for hand evaluation
* This changes the effective suit for evaluation, especially for flushes
*/
Card transform_card_suit_bundler(Card card)
{
    if (joker_cards[3] == 0) { // Check if this joker is not active
        return card;
    }

    if (card == 0 || card == 0xFF) return card; // Skip invalid cards

    int rank = RANK(card);
    int suit = SUIT(card);

    // Transform suits: Spades(2) -> Clubs(0), Diamonds(1) -> Hearts(3)
    if (suit == 2) { // Spades -> Clubs
        return MAKE_CARD(rank, 0);
    } else if (suit == 1) { // Diamonds -> Hearts
        return MAKE_CARD(rank, 3);
    }

    return card; // No change for clubs and hearts
}

/*
```

```
* Apply Suit Bundler to an entire hand
* Used before hand evaluation to transform all cards
*/
void apply_suit_bundler_joker(Card *hand, int num_cards, Card
*transformed_hand)
{
   for (int i = 0; i < num_cards; i++) {
       transformed_hand[i] = transform_card_suit_bundler(hand[i]);
   }
}

/*
* The One (Legendary) ~ Next 7 Hands are Royal Flushes
* Sets a counter to transform the next 7 hands into royal flushes
*/
int the_one_hands_remaining = 0;

void activate_the_one_joker()
{
   if (joker_cards[4] == 0) { // Check if this joker is not active
       return;
   }

   the_one_hands_remaining = 7;
   joker_cards[4] = 0; // Consume the joker once activated
}

/*
* Check if The One should transform the current hand
* Returns 1 if the hand should be treated as a royal flush
*/
int check_the_one_joker()
{
   if (the_one_hands_remaining > 0) {
       the_one_hands_remaining--;
       return 1; // Treat as royal flush
   }
   return 0;
}

/*
* Blue Dot (Common) ~ Plus 50 Chip
* Adds a flat 50 chips to any hand
*/
```

```
int apply_blue_dot_joker(int base_chips)
{
   if (joker_cards[5] == 0) { // Check if this joker is not active
       return base_chips;
   }

   return base_chips + 50;
}

/*
* Red Dot (Common) ~ Plus 10 Mult
* Adds a flat 10 to the multiplier of any hand
*/
int apply_red_dot_joker(int base_multiplier)
{
   if (joker_cards[6] == 0) { // Check if this joker is not active
       return base_multiplier;
   }

   return base_multiplier + 10;
}

/*
* Green Check (Rare) ~ Need Only 25 Percent to Pass Round (One Time Use but
will
* activate when no hands left)
* Reduces target score to 25% when no hands are left and the target wasn't
reached
*/
int green_check_used = 0;
int apply_green_check_joker(int target_score, int current_score, int
hands_left)
{
   if (joker_cards[7] == 0 || green_check_used) { // Check if joker inactive or
already used
       return 0; // Not applicable
   }

   // Activate when no hands left and score is below target
   if (hands_left == 0 && current_score < target_score) {
       int reduced_target = target_score / 4; // 25% of original target

       // If player has reached the reduced target
       if (current_score >= reduced_target) {
```

```
            green_check_used = 1; // Mark as used
            joker_cards[7] = 0;   // Deactivate the joker
            return 1; // Indicate success
        }
    }

    return 0; // Not activated yet
}

/*
* Stevie Dott (Rare) ~ Even Sum Get 40 Mult Odd Sum Get 2X On Played Card Chip
* Changes multiplier or doubles card chips based on sum of card ranks
*/
void apply_stevie_dott_joker(Card *hand, int num_cards, int *multiplier, int
*card_chips)
{
    if (joker_cards[8] == 0) { // Check if this joker is not active
        return;
    }

    // Calculate sum of card ranks
    int rank_sum = 0;
    for (int i = 0; i < num_cards; i++) {
        rank_sum += RANK(hand[i]);
    }

    if (rank_sum % 2 == 0) { // Even sum
        *multiplier = 40; // Set multiplier to 40
    } else { // Odd sum
        *card_chips *= 2; // Double the card chips
    }
}

/*
* Helper function to apply all active joker effects to a hand's score
* This would be called after normal hand evaluation
*/
void apply_all_joker_effects(Card *hand, int num_cards, enum HandType
*hand_type,
                             int *chips, int *multiplier, int target_score,
                             int current_score, int hands_left)
{
    // Make a copy of the hand for potential transformations
    Card transformed_hand[5] = {0};
```

```
    memcpy(transformed_hand, hand, num_cards * sizeof(Card));

    // Apply suit bundler transformation
    if (joker_cards[3]) {
        apply_suit_bundler_joker(hand, num_cards, transformed_hand);
        // Re-evaluate hand with transformed suits
        *hand_type = evaluate_hand(transformed_hand, num_cards);
    }

    // Apply The One (royal flush override)
    if (check_the_one_joker()) {
        *hand_type = ROYAL_FLUSH;
    }

    // Get base hand value after possible transformations
    HandValue hand_value = get_hand_value(*hand_type);
    *chips = hand_value.chips;
    *multiplier = hand_value.multiplier;

    // Apply joker effects that modify chips
    *chips = apply_odd_todd_joker(hand, num_cards, *chips);
    *chips = apply_blue_dot_joker(*chips);

    // Apply joker effects that modify multiplier
    *multiplier = apply_smiley_face_joker(hand, num_cards, *multiplier);
    *multiplier = apply_even_steven_joker(hand, num_cards, *multiplier);
    *multiplier = apply_red_dot_joker(*multiplier);

    // Calculate card-specific chips
    int hand_specific_chips = calculate_hand_type_specific_chips();
    *chips += hand_specific_chips;

    // Apply Stevie Dott effect (may modify multiplier or hand_specific_chips)
    apply_stevie_dott_joker(hand, num_cards, multiplier, &hand_specific_chips);

    // Check if Green Check should be applied
    apply_green_check_joker(target_score, current_score, hands_left);
}

/*
* GAME LOOP
*/
void game_loop_vga() {
    struct libusb_device_handle *controller;
```

```
uint8_t ep;
unsigned char report[REPORT_LEN];
int transferred;

// Initialize controller
controller = opencontroller(&ep);
if (!controller) {
    printf("Failed to open controller\n");
    return;
}

while (!game_play) {
    // Input processing
    int r = libusb_interrupt_transfer(controller, ep,
        report, REPORT_LEN,
        &transferred, 100); // 100ms timeout

    if (r == 0 && transferred > 0) {
        process_controller_input(report);
    }

}

// Initialize debounce
initialize_debounce();

// Set up initial game state
hard_reset();

// Initialize joker display (empty)
uint8_t empty_jokers[5] = {10, 10, 10, 10, 10}; // 10 is the code for empty
draw_jokers(empty_jokers);

draw_hand(index_for_tiles_cards);

// Get initial game info
char blind_name[20];
int target_score;
int cumulative_score = 0;
get_current_blind_info(blind_name, &target_score);

// Initialize VGA display with game state
draw_target_score(get_target_score_vga(target_score));
draw_round_score(get_round_score_vga(cumulative_score));
```

```
draw_hand_type("POKER HAND TYPE");
draw_chip(get_chip_vga(0));
draw_mult(get_mult_vga(0));
draw_hands_left(get_hands_left_vga()[0]);
draw_discards(get_discards_left_vga()[0]);
draw_cards_in_deck(get_cards_in_deck_vga());
draw_ante(get_ante_vga()[0]);
draw_round(get_round_vga()[0]);
draw_selected(get_selected_cards_array());

// Draw initial cursor
cursor_position = get_cursor_position();
draw_cursor(cursor_position);

while (game_play) {

    // Input processing
    int r = libusb_interrupt_transfer(controller, ep,
                                report, REPORT_LEN,
                                &transferred, 100); // 100ms timeout

    if (r == 0 && transferred > 0) {
        // Save previous game state for change detection
        update_clear_cursor_position(); // Save current cursor for clearing
        int previous_hands = hands_remaining;
        int previous_discards = discards_remaining;
        int previous_num_selected = num_selected_cards;

        // Process controller input
        int old_button = last_button;
        process_controller_input(report);

        // Get current cursor position and update cursor display if it moved
        cursor_position = get_cursor_position();
        clear_cursor_position = get_clear_cursor_position();

        if (cursor_position != clear_cursor_position) {
            clear_cursor(clear_cursor_position);
            draw_cursor(cursor_position);
        }

        // If selections changed, update hand evaluation display
        if (num_selected_cards != previous_num_selected) {
            // Get current hand evaluation
```

```
                enum HandType hand_type = evaluate_selected_cards();
                HandValue hand_value = get_hand_value(hand_type);

                // Update the display
                draw_hand_type((char*)get_hand_name_vga(hand_type));
                draw_chip(get_chip_vga(hand_value.chips));
                draw_mult(get_mult_vga(hand_value.multiplier));

                // Update selected cards display
                draw_selected(get_selected_cards_array());
            }
            if (num_selected_cards == 0) {
                draw_hand_type("POKER HAND TYPE");
                draw_chip(get_chip_vga(0));
                draw_mult(get_mult_vga(0));
            }

            // Check if a hand was just played (START button pressed)
            if (last_button == 8 && old_button != 8 && previous_num_selected > 0
&&
                hands_remaining < previous_hands) {

                // Calculate the score for the hand that was just played
                enum HandType hand_type = evaluate_hand(played_hand,
previous_num_selected);
                HandValue hand_value = get_hand_value(hand_type);

                // Calculate card-specific chips using the played hand instead
of selected_cards
                int hand_specific_chips =
calculate_played_hand_chips(played_hand, previous_num_selected, hand_type);

                // Apply joker effects to the hand
                int modified_chips = hand_value.chips;
                int modified_multiplier = hand_value.multiplier;

                // Apply all active joker effects to the hand
                apply_all_joker_effects(
                    played_hand,
                    previous_num_selected,
                    &hand_type,  // Hand type might be upgraded by jokers
                    &modified_chips,
                    &modified_multiplier,
                    target_score,
```

```
                    cumulative_score,
                    hands_remaining
                );

                // Use the modified values for scoring
                int total_chip_value = modified_chips + hand_specific_chips;
                int score = total_chip_value * modified_multiplier;

                // Add score to running total
                cumulative_score += score;

                // Display played hand on the VGA
                uint8_t played_card_indices[5] = {52, 52, 52, 52, 52}; //
Initialize all to "empty"
                for (int i = 0; i < previous_num_selected; i++) {
                    // Convert each card to its deck index
                    int rank = RANK(played_hand[i]);
                    int suit = SUIT(played_hand[i]);
                    played_card_indices[i] = suit * 13 + rank;
                }

                draw_played_cards(played_card_indices);

                // Create a temporary array to modify for display
                uint8_t updated_hand_indices[8];
                for (int i = 0; i < draw_amount; i++) {
                    updated_hand_indices[i] = index_for_tiles_cards[i];
                }

                // Mark played cards as empty (52) in the display array
                for (int i = 0; i < 5; i++) {
                    for (int j = 0; j < 8; j++) {
                        if (played_card_indices[i] == index_for_tiles_cards[j])
{
                            updated_hand_indices[j] = 52; // Set to empty
                            break;
                        }
                    }
                }
                // Update the hand display with empty slots for played cards
                draw_hand(updated_hand_indices);

                // Update the VGA display with new score and game state
                draw_round_score(get_round_score_vga(cumulative_score));
```

```
                draw_hands_left(get_hands_left_vga()[0]);
                draw_cards_in_deck(get_cards_in_deck_vga());

                // Also show the hand type that was just played
                draw_hand_type((char*)get_hand_name_vga(hand_type));
                draw_chip(get_chip_vga(total_chip_value));
                draw_mult(get_mult_vga(modified_multiplier));

                // Pause briefly to show the played hand
                sleep(2);
                clear_table((uint8_t) previous_num_selected);

                // Draw replacement cards
                draw_replacement_cards();
                drawn_to_index();
                draw_hand(index_for_tiles_cards);

                // Reset cursor position
                clear_cursor(cursor_position);
                cursor = 0;
                cursor_position = get_cursor_position();
                draw_cursor(cursor_position);

                // Reset selected cards display
                draw_selected(get_selected_cards_array());

                // Check win condition
                if (check_win_condition(cumulative_score)) {
                    // Display win message
                    draw_hand_type("  YOU WIN    ");
                    sleep(2);
                    // Award a New Possible Joker
                    joker_drop();

                    // Refresh joker display
                    uint8_t joker_display[5] = {10, 10, 10, 10, 10}; //
Initialize empty
                    int joker_count = 0;
                    for (int i = 0; i < 10 && joker_count < 5; i++) {
                        if (joker_cards[i] == 1) {
                            joker_display[joker_count++] = i;
                        }
                    }
                    draw_jokers(joker_display);
```

```
                    // Advance to next blind/ante
                    advance_game_state();
                    if (game_play) {
                        cumulative_score = 0;

                        // Update display for new game state
                        get_current_blind_info(blind_name, &target_score);
                        draw_target_score(get_target_score_vga(target_score));
                        draw_round_score(get_round_score_vga(cumulative_score));
                        draw_hands_left(get_hands_left_vga()[0]);
                        draw_discards(get_discards_left_vga()[0]);
                        draw_cards_in_deck(get_cards_in_deck_vga());
                        draw_ante(get_ante_vga()[0]);
                        draw_round(get_round_vga()[0]);

                        // Reset hand evaluation display
                        draw_hand_type("POKER HAND TYPE");
                        draw_chip(get_chip_vga(0));
                        draw_mult(get_mult_vga(0));

                        // Draw new hand
                        drawn_to_index();
                        draw_hand(index_for_tiles_cards);

                        // Reset cursor and selection display
                        clear_cursor(cursor_position);
                        cursor = 0;
                        cursor_position = get_cursor_position();
                        draw_cursor(cursor_position);
                        draw_selected(get_selected_cards_array());
                    }
                }
            }

            // Check if cards were discarded (SELECT button pressed)
            if (last_button == 7 && old_button != 7 && previous_discards >
discards_remaining) {
                // Draw replacement cards
                draw_replacement_cards();
                drawn_to_index();
                draw_hand(index_for_tiles_cards);

                // Update display
```

```
                draw_discards(get_discards_left_vga()[0]);
                draw_cards_in_deck(get_cards_in_deck_vga());

                // Reset hand evaluation display
                draw_hand_type("POKER HAND TYPE");
                draw_chip(get_chip_vga(0));
                draw_mult(get_mult_vga(0));

                // Reset cursor and selection display
                clear_cursor(cursor_position);
                cursor = 0;
                cursor_position = get_cursor_position();
                draw_cursor(cursor_position);
                draw_selected(get_selected_cards_array());
            }
        }

        // Check if game is over (out of hands and didn't win)
        if (hands_remaining == 0) {
            // Check if Green Check would save us (reduces target to 25%)
            if (joker_cards[7] == 1 && !green_check_used) {
                // Get current target score
                get_current_blind_info(blind_name, &target_score);
                int reduced_target = target_score / 4;

                if (cumulative_score >= reduced_target) {
                    // Green Check activates!
                    joker_cards[7] = 0; // Consume the joker
                    green_check_used = 1;

                    // Display Green Check activation message
                    draw_hand_type("GREEN CHECK WIN!");
                    sleep(2);

                    // Update joker display to remove green check
                    uint8_t joker_display[5] = {10, 10, 10, 10, 10}; //
Initialize empty
                    int joker_count = 0;
                    for (int i = 0; i < 10 && joker_count < 5; i++) {
                        if (joker_cards[i] == 1) {
                            joker_display[joker_count++] = i;
                        }
                    }
                    draw_jokers(joker_display);
```

```
            // Award a possible new joker
            joker_drop();

            // Refresh joker display
            joker_count = 0;
            for (int i = 0; i < 10 && joker_count < 5; i++) {
                if (joker_cards[i] == 1) {
                    joker_display[joker_count++] = i;
                }
            }
            draw_jokers(joker_display);

            // Handle win as if target was reached
            advance_game_state();
            if (game_play) {
                cumulative_score = 0;

                // Update display for new game state
                get_current_blind_info(blind_name, &target_score);
                draw_target_score(get_target_score_vga(target_score));
                draw_round_score(get_round_score_vga(cumulative_score));
                draw_hands_left(get_hands_left_vga()[0]);
                draw_discards(get_discards_left_vga()[0]);
                draw_cards_in_deck(get_cards_in_deck_vga());
                draw_ante(get_ante_vga()[0]);
                draw_round(get_round_vga()[0]);

                // Reset hand evaluation display
                draw_hand_type("POKER HAND TYPE");
                draw_chip(get_chip_vga(0));
                draw_mult(get_mult_vga(0));

                // Draw new hand
                drawn_to_index();
                draw_hand(index_for_tiles_cards);
            }
        }
        else if (!check_win_condition(cumulative_score)) {
            // Game over logic
            draw_hand_type("  GAME OVER   ");
            sleep(1);
            // CALL GAME OVER
            game_over();
```

```
                }
            }
            else if (!check_win_condition(cumulative_score)) {
                // Normal game over without Green Check
                draw_hand_type("  GAME OVER   ");
                sleep(1);
                // CALL GAME OVER
                game_over();
            }
        }

        // Small delay to prevent maxing out the CPU
        usleep(10000); // 10ms delay
    }

    // Clean up
    libusb_close(controller);
    libusb_exit(NULL);
}

/*
* Main Method
*/
int main()
{
    static const char filename[] = "/dev/vga_poker";
    printf("VGA tiles Userspace program started\n");
    if ( (vga_poker_fd = open(filename, O_RDWR)) == -1) {
      fprintf(stderr, "could not open %s\n", filename);
      return -1;
    }

    /* 1) load all the on-disk graphics into FPGA memory */
    load_palette("palette.hex");
    load_tileset("tileset.hex");
    load_tilemap("256-title.hex");
    load_deck("256-cards.hex");
    load_jokers("256-Jokers.hex");

    while(1) {
        game_loop_vga();
        load_tilemap("256-title.hex");
    }
```

```
   close(vga_poker_fd);
   return 0;
}
```

controller.h

```
C/C++
#ifndef _CONTROLLER_H
#define _CONTROLLER_H

#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

struct usb_keyboard_packet {
 uint8_t modifiers;
 uint8_t reserved;
 uint8_t keycode[6];
};

/* Find and open a USB controller device.  Argument should point to
  space to store an endpoint address.  Returns NULL if no controller
  device was found. */
extern struct libusb_device_handle *opencontroller(uint8_t *);

#endif
```

controller.c

```
C/C++
#include "controller.h"
#include <libusb-1.0/libusb.h>
#include <stdio.h>
#include <stdlib.h>

/* Vendor / product IDs for the DragonRise Inc. USB Gamepad          */
#define DRAGONRISE_VID  0x0079
#define DRAGONRISE_PID  0x0011

struct libusb_device_handle *opencontroller(uint8_t *endpoint_address)
{
   libusb_device       **devs;
```

```
struct libusb_device_handle *handle = NULL;
struct libusb_device_descriptor desc;
ssize_t              ndevs;

if (libusb_init(NULL) < 0) {
    perror("libusb_init");
    return NULL;
}

ndevs = libusb_get_device_list(NULL, &devs);
if (ndevs < 0) {
    perror("libusb_get_device_list");
    return NULL;
}

/* enumerate every device ---------------------------------------- */
for (ssize_t d = 0; d < ndevs; ++d) {
    libusb_device *dev = devs[d];

    if (libusb_get_device_descriptor(dev, &desc) != 0)
        continue;

    if (desc.idVendor  != DRAGONRISE_VID ||
        desc.idProduct != DRAGONRISE_PID)
        continue;                                   /* not our pad   */

    /* walk its interfaces ------------------------------------------ */
    struct libusb_config_descriptor *cfg;
    libusb_get_config_descriptor(dev, 0, &cfg);

    for (uint8_t i = 0; i < cfg->bNumInterfaces; ++i)
        for (uint8_t a = 0; a < cfg->interface[i].num_altsetting; ++a) {

            const struct libusb_interface_descriptor *ifd =
                &cfg->interface[i].altsetting[a];

            if (ifd->bInterfaceClass    == LIBUSB_CLASS_HID &&
                ifd->bInterfaceSubClass == 0x00 &&          /* generic */
                ifd->bInterfaceProtocol == 0x00) {          /* none    */

                if (libusb_open(dev, &handle) != 0)
                    break;

                if (libusb_kernel_driver_active(handle, i))
```

```
                    libusb_detach_kernel_driver(handle, i);

                libusb_claim_interface(handle, i);

                *endpoint_address = ifd->endpoint[0].bEndpointAddress;
                libusb_free_config_descriptor(cfg);
                libusb_free_device_list(devs, 1);
                return handle;                  /* success! */
            }
        }

    libusb_free_config_descriptor(cfg);
  }

  libusb_free_device_list(devs, 1);
  libusb_exit(NULL);
  return NULL;                                  /* not found */
}
```

tiles.sv

```
Unset
module tiles
  (input logic         VGA_CLK, VGA_RESET,
   output logic [7:0]  VGA_R, VGA_G, VGA_B,
   output logic        VGA_HS, VGA_VS, VGA_BLANK_n,

   input logic         mem_clk,         // Clock for memory ports

   input logic [12:0]  tm_address,      // Tilemap memory port
   input logic         tm_we,
   input logic [7:0]   tm_din,
   output logic [7:0]  tm_dout,

   input logic [13:0]  ts_address,      // Tileset memory port
   input logic         ts_we,
   input logic [3:0]   ts_din,
   output logic [3:0]  ts_dout,

   input logic [3:0]   palette_address, // Palette memory port
   input logic         palette_we,
   input logic [23:0]  palette_din,
```

```
output logic [23:0] palette_dout);

logic [9:0]         hcount;          // From counters
logic [8:0]         vcount;

logic [2:0]         hcount1;         // Pipeline registers
logic               VGA_HS0, VGA_HS1, VGA_HS2;
logic               VGA_BLANK_n0, VGA_BLANK_n1, VGA_BLANK_n2;

logic [7:0]         tilenumber;      // Memory outputs
logic [3:0]         colorindex;

/* verilator lint_off UNUSED */
logic               unconnected; // Extra vcount bit from counters
/* verilator lint_on UNUSED */

vga_counters cntrs(.vcount( {unconnected, vcount} ), // VGA Counters
                   .VGA_BLANK_n( VGA_BLANK_n0 ),
                   .VGA_HS( VGA_HS0 ),
                   .*);

twoportbram #(.DATA_BITS(8), .ADDRESS_BITS(13))  // Tile Map
tilemap(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
        .addr1 ( { vcount[8:3], hcount[9:3] } ),
        .we1   ( 1'b0 ), .din1( 8'h X ), .dout1( tilenumber ),
        .addr2 ( tm_address ),
        .we2   ( tm_we ), .din2( tm_din ), .dout2( tm_dout ));

always_ff @(posedge VGA_CLK)                     // Pipeline registers
  { hcount1, VGA_BLANK_n1, VGA_HS1 } <=
    { hcount[2:0], VGA_BLANK_n0, VGA_HS0 };

twoportbram #(.DATA_BITS(4), .ADDRESS_BITS(14))  // Tile Set
tileset(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
        .addr1 ( { tilenumber, vcount[2:0], hcount1 } ),
    .we1   ( 1'b0 ), .din1( 4'h X), .dout1( colorindex ),
        .addr2 ( ts_address ),
        .we2   ( ts_we ), .din2( ts_din ), .dout2( ts_dout ));

always_ff @(posedge VGA_CLK)                     // Pipeline registers
  { VGA_BLANK_n2, VGA_HS2 } <= { VGA_BLANK_n1, VGA_HS1 };

twoportbram #(.DATA_BITS(24), .ADDRESS_BITS(4))  // Palette
palette(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
```

```
            .addr1 ( colorindex ),
            .we1   ( 1'b0 ), .din1( 24'h X ), .dout1( { VGA_B, VGA_G, VGA_R } ),
            .addr2 ( palette_address ),
            .we2   ( palette_we ), .din2( palette_din ), .dout2( palette_dout
));

   always_ff @(posedge VGA_CLK)                    // Pipeline registers
     { VGA_BLANK_n, VGA_HS } <= { VGA_BLANK_n2, VGA_HS2 };

endmodule
```

twoportbram.sv

```
Unset
module twoportbram
  #(parameter int DATA_BITS = 8,  ADDRESS_BITS = 10)
   (input logic                    clk1,  clk2,
    input logic [ADDRESS_BITS-1:0] addr1, addr2,
    input logic [DATA_BITS-1:0]    din1,  din2,
    input logic                    we1,   we2,
    output logic [DATA_BITS-1:0]   dout1, dout2);

   localparam WORDS = 1 << ADDRESS_BITS;

   /* verilator lint_off MULTIDRIVEN */
   logic [DATA_BITS-1:0]           mem [WORDS-1:0];
   /* verilator lint_on MULTIDRIVEN */

   always_ff @(posedge clk1)
     if (we1) begin
        mem[addr1] <= din1;
        dout1 <= din1;
     end else dout1 <= mem[addr1];

   always_ff @(posedge clk2)
     if (we2) begin
        mem[addr2] <= din2;
        dout2 <= din2;
     end else dout2 <= mem[addr2];

endmodule
```

vga_counters.sv

```
Unset
module vga_counters(
   input  logic       VGA_CLK, VGA_RESET,
   output logic [9:0] hcount, // 0-639 active, 640-799 blank/sync
   output logic [9:0] vcount, // 0-479 active, 480-524 blank/sync
   output logic       VGA_HS, VGA_VS, VGA_BLANK_n);

   logic endOfLine;
   assign endOfLine = hcount == 10'd 799;

   always_ff @(posedge VGA_CLK or posedge VGA_RESET)
     if (VGA_RESET)      hcount <= 10'd 797;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 10'd 1;

   logic endOfFrame;
   assign endOfFrame = vcount == 10'd 524;

   always_ff @(posedge VGA_CLK or posedge VGA_RESET)
     if (VGA_RESET)      vcount <= 10'd 524;
     else if (endOfLine)
       if (endOfFrame)   vcount <= 10'd 0;
       else              vcount <= vcount + 10'd 1;

   // 656 <= hcount <= 751
   assign VGA_HS = !( hcount[9:7] == 3'b101 &
                      hcount[6:4] != 3'b000 & hcount[6:4] != 3'b111 );
   assign VGA_VS = !( vcount[9:1] == 9'd 245 ); // Lines 490 and 491

   // hcount < 640 && vcount < 480
   assign VGA_BLANK_n = !( hcount[9] & (hcount[8] | hcount[7]) ) &
                        !( vcount[9] | (vcount[8:5] == 4'b1111) );
endmodule
```

vga_tiles.sv

```
Unset
/*
 * Avalon memory-mapped agent peripheral that produces a VGA tile display
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Memory map:
```

```
 *
 * 0000 - 1FFF Tilemap (8K, tile number is 8 bits per byte)
 * 2000 - 203F Palette (64, 24 bits every 4 bytes)
 * 4000 - 7FFF Tileset (16K, color index is lower 4 bits of each byte)
 *
 * 00m mmmm mmmm mmmm  Tilemap
 * 010 0000 00pp ppbb  Palette
 * 1ss ssss ssss ssss  Tileset
 *
 * In the 64-byte palette region, every color occupies 4 bytes, although
 * only 24 bits are stored.  Writing to the first 3 bytes in each group
 * writes a byte into the 24-bit color register.  Writing to the fourth
 * byte writes the color register to the palette memory; any data written to
 * these addresses is ignored; they always read 0.
 *
 * | Offset |    On Write          |  On Read          |
 * +--------+----------------------+-------------------+
 * |   0    | creg[7:0] <- data    | palette[0].red    |
 * |   1    | creg[15:8] <- data   | palette[0].green  |
 * |   2    | creg[23:16] <- data  | palette[0].blue   |
 * |   3    | palette[0] <- creg   | Always 0          |
 * |   4    | creg[7:0] <- data    | palette[1].red    |
 * |   5    | creg[15:8] <- data   | palette[1].green  |
 * |   6    | creg[23:16] <- data  | palette[1].blue   |
 * |   7    | palette[1] <- creg   | Always 0          |
 *  ...
 * |   60    | creg[7:0] <- data   | palette[15].red   |
 * |   61    | creg[15:8] <- data  | palette[15].green |
 * |   62    | creg[23:16] <- data | palette[15].blue  |
 * |   63    | palette[15] <- creg | Always 0          |
 *
 */
module vga_tiles
  (input logic        clk, reset,                    // Avalon MM Agent port
   input logic        chipselect, write,             // read == chipselect &
!write
   input logic [14:0] address,                       // 32K window
   input logic [7:0]  writedata,                     // 8-bit interface
   output logic [7:0] readdata,

   input logic        vga_clk_in, VGA_RESET,         // VGA signals
   output logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n);
```

```
   logic [2:0]       creg_write;                    // Latch enable per byte
   logic             tm_we, ts_we, palette_we;      // Memory write enables
   logic [7:0]       tm_dout;                       // Data from tilemap
   logic [3:0]       ts_dout;                       // Data from tileset
   logic [23:0]      creg, palette_dout;            // Data to/from palette

   tiles tiles(.mem_clk       ( clk           ),
               .tm_address    ( address[12:0] ), .tm_din     ( writedata
),
               .ts_address    ( address[13:0] ), .ts_din     ( writedata[3:0]
),
               .palette_address( address[5:2]  ), .palette_din( creg
), .*);
   assign VGA_CLK = vga_clk_in;

   always_comb begin                                    // Address Decoder
      {tm_we, ts_we, palette_we, creg_write, readdata } = { 6'b 0, 8'h xx };
      if (chipselect)
        if (address[14] == 1'b 1) begin                 // Tileset
1--------------
          ts_we    = write;                           //  Write to tileset mem
          readdata = { 4'h 0, ts_dout };              //  Read lower 4 bits;
pad upper
        end else if (address[13] == 1'b 0) begin        // Tilemap
00-------------
          tm_we    = write;                           //  Write to tilemap mem
          readdata = tm_dout;                         //  Read 8 bits
        end else if ( address[12:6] == 7'b 0_0000_00 ) // Palette
010000000------
          case (address[1:0])
            2'h 0 : begin readdata = palette_dout[7:0];   // Read red byte
                          creg_write[0] = write;          // creg <- red
                    end
            2'h 1 : begin readdata = palette_dout[15:8];  // Read green byte
                          creg_write[1] = write;          // creg <- green
                    end
            2'h 2 : begin readdata = palette_dout[23:16]; // Read blue byte
                          creg_write[2] = write;          // creg <- blue
                    end
            2'h 3 : begin readdata = 8'h 00;              // Always reads as
00
                         palette_we = write;             // mem <- creg
                    end
          endcase
```

```
    end

    always_ff @(posedge clk or posedge reset)
      if (reset) creg <= 24'b 0; else begin
         if (creg_write[0]) creg[7:0]   <= writedata;    // Write byte (color)
         if (creg_write[1]) creg[15:8]  <= writedata;    // to creg according to
         if (creg_write[2]) creg[23:16] <= writedata;    // creg_write bits
      end
endmodule
```