



PROJECT REPORT: ARCADE KERNEL v4.2

High-Performance Embedded Management System

Department: Computer Science & Engineering

University: University of Petroleum and Energy Studies (UPES)

GROUP MEMBERS

Name	Sap ID
Raghav Sharma	590028524
Praneeth Sai Varada	590028521

1. PROBLEM DEFINITION

1.1 Context of the Computational Problem

In the domain of embedded entertainment systems and high-throughput transaction processing, legacy management systems rely heavily on manual token counting and static pricing models. This creates operational inefficiencies, including revenue leakage, lack of real-time user data, and hardware downtime due to unmonitored "zombie" processes.

Just as environmental sustainability requires resource stewardship, computational sustainability requires **efficient memory management**, **latency-free execution**, and **robust error handling**. This project addresses the critical need for a centralized, automated operating environment (Kernel) to manage distributed hardware resources.

1.2 Objective of the Development

The primary objective of this project is to architect a C-language-based "**Arcade Kernel**" that translates theoretical Data Structure concepts into a tangible, executable application. This utility aims to:

- **Automate Identity Management:** Replace physical tokens with an RFID-based digital ledger using **Hash Maps** to ensure $\$O(1)$ access time.
- **Optimize Revenue:** Implement a "**Surge Pricing**" **algorithm** that dynamically adjusts interaction costs based on real-time machine load.
- **Ensure Stability:** Deploy a "**Watchdog Timer**" to detect and recover stuck processes without human intervention, ensuring high availability.
- **Guarantee Persistence:** Implement binary serialization to secure user credits and tickets against power loss.

1.3 Scope and Constraints

The solution is designed to operate within a standard GCC compiler environment (C99 Standard), focusing on modularity and code reusability. Constraints included strict memory optimization (preventing leaks via manual heap management) and the use of low-level file I/O for data security.

2. ALGORITHMS AND DATA STRUCTURES

2.1 Core Data Structure: Hash Table (Chaining)

File: player_manager.c

To ensure instant user lookup during RFID scans, we implemented a Hash Table with Chaining.

- **Algorithm:** The hash_func utilizes a standard string hashing algorithm (djb2 variant), bit-shifting the hash value by 5 bits ($\text{hash} \ll 5$) to ensure a uniform distribution across the 1024-slot table.
- **Collision Handling:** Linked Lists are used to handle collisions. If two RFID tags generate the same index, the new player is appended to the linked list node (`current->next`).

2.2 Algorithm: Dynamic Surge Pricing

File: machine_manager.c

The system moves beyond static variables by implementing a supply-demand algorithm.

- **Logic:** The system monitors session_plays. If a machine exceeds a specific threshold (10 plays), the STATUS_HIGH_LOAD bitmask is triggered.
- **Mathematical Model:** $\text{Cost}_{\{\text{new}\}} = \text{Cost}_{\{\text{base}\}} \times 1.5$
- **Impact:** This maximizes virtual revenue during peak traffic periods automatically.

2.3 Algorithm: The Watchdog Timer

File: machine_manager.c

To emulate real-time embedded systems, a Watchdog mechanism was engineered.

1. **Traversal:** The function run_watchdog_scan iterates through the hardware bus.
2. **Time Delta:** It calculates `difftime(now, m->last_heartbeat)`.
3. **Recovery:** If the delta > 120 seconds while the status is PLAYING, the machine is forcibly reset to IDLE and flagged as JAMMED.

2.4 Data Structure: Circular Buffer (Ghost Logs)

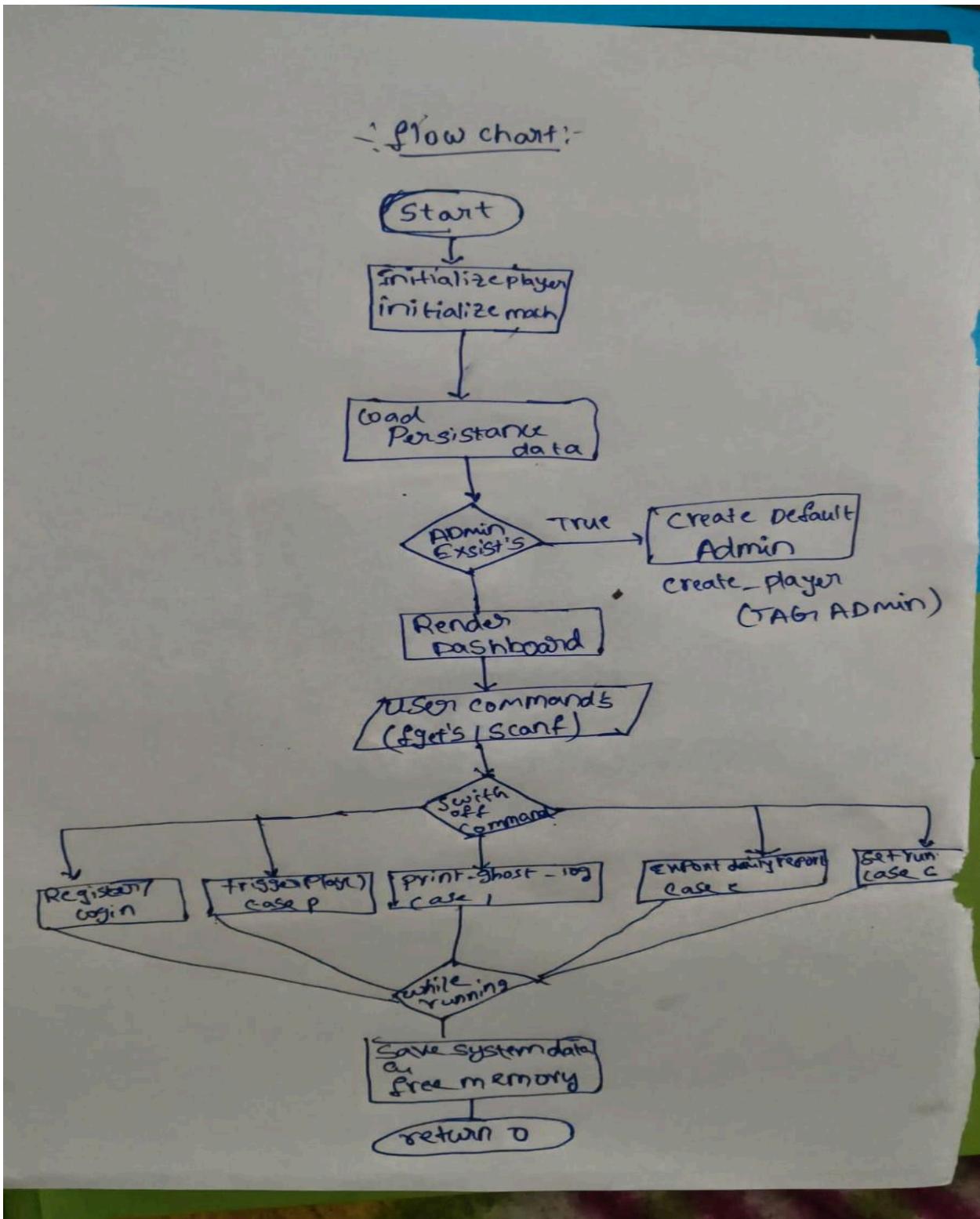
File: core_types.h

For debugging without consuming infinite memory, each machine maintains a fixed-size ActionLog `logs[GHOST_LOG_SIZE]`.

- Implementation: The `log_head` index increments using the modulo operator:
$$\text{Index} = (\text{Current} + 1) \% \text{Size}$$

This ensures that old logs are automatically overwritten by new events, maintaining a constant memory footprint.

3. FLOWCHART



4. PROBLEMS FACED BY TEAM MEMBERS

During our code development life-cycle, the team encountered specific technical hurdles related to memory management, I/O streams, and data persistence. These issues were documented and resolved as follows:

4.1 Memory Segmentation Faults

- **Issue:** Early versions of the `find_player` function caused the kernel to crash ("Segmentation Fault") when traversing the linked list. This occurred when the system attempted to access `current->next` on a pointer that was effectively NULL.
- **Resolution:** We implemented strict pointer safety protocols. We ensured the entire `player_table` was initialized to zero using `memset` during boot. Additionally, we added conditional guards (`(!current) return NULL;`) before any pointer dereferencing occurred.

4.2 Input Stream Corruption ("The Ghost Enter Key")

- **Issue:** During user interaction, the system would often skip input prompts (e.g., skipping the "Enter Name" step after scanning a tag). This was caused by `scanf` leaving newline characters (`\n`) in the standard input buffer, which the subsequent `fgets` would consume as an empty command.
- **Resolution:** We engineered a dedicated helper function, `flush_input()`, which consumes all characters in the buffer until a newline or EOF is reached. This function is called immediately after every `scanf` operation to ensure a clean input stream.

4.3 Data Serialization (Persistence)

- **Issue:** Saving the state of the system was complex because the `Player` struct contains pointers (`struct Player *next`). We realized that memory addresses (pointers) cannot be saved to a binary file as they become invalid upon restarting the program.
- **Resolution:** We devised a "Flattening" strategy. Instead of saving the `Player` struct directly, we created a temporary intermediate structure, `PlayerFileRecord`, which contains only static data types (arrays, ints). The system converts the dynamic Linked List into these flat records before writing to `arcade_data.bin`.

5. APPENDIX A: SOURCE CODE

5.1 MakeFile

```
# Compiler Configuration
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -O2
TARGET = arcade_os

# Source definitions
SRCS = main.c player_manager.c machine_manager.c ui_renderer.c
persistence_manager.c
OBJS = $(SRCS:.c=.o)
# Simple dependency tracking
DEPS = core_types.h persistence.h

.PHONY: all clean rebuild debug

# Default build target
all: $(TARGET)

# Linking phase
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Explicit Compilation Rule
%.o: %.c $(DEPS)
    $(CC) $(CFLAGS) -c -o $@ $<

debug: CFLAGS = -Wall -Wextra -std=c99 -g -O0
debug: clean $(TARGET)

# Clean up build artifacts
clean:
    rm -f $(OBJS) $(TARGET) *.bin *.csv

# Rebuild from scratch
rebuild: clean all
```

5.2 core_types.h

```
#ifndef CORE_TYPES_H
#define CORE_TYPES_H


#include<time.h>
#include<stdint.h>
#include<stdbool.h>

// ---SYSTEM CONSTANTS---

#define MAX_PLAYERS_HASH 1024 // Size of Hash Table
#define MAX_MACHINES 50 // Max physical cabinets supported
#define GHOST_LOG_SIZE 20 // Size of the circular buffer for
debugging
#define ID_LEN 16 // Length of RFID Tag Strings

// ---BITMASKS FOR MACHINE STATUS---

#define STATUS_OFFLINE 0x00
#define STATUS_IDLE 0x01
#define STATUS_PLAYING 0x02
#define STATUS_JAMMED 0x04
#define STATUS_HIGH_LOAD 0x08

// ---DATA STRUCTURES---


// 1. Ghost Maintenance Log
typedef struct{
    time_t timestamp;
    char action_type; // 'P'lay, 'E'rror, 'M'aintenance
    int value;
} ActionLog;

// 2. Machine Logic Unit
typedef struct{
    int id;
    char name[32];
    uint8_t flags;
    float cost_per_play;
}
```

```
    float base_cost;
    int total_plays;
    int session_plays;

    time_t last_heartbeat;

    // Competitive Data
    int high_score;
    char champion_name[32];

    // The "Black Box" circular buffer
    ActionLog logs[GHOST_LOG_SIZE];
    int log_head;
}Machine;

// 3. Player Account (Hash Table Node)
typedef struct Player{
    char rfid_tag[ID_LEN];
    char name[32];
    int credits;
    int tickets;
    time_t last_seen;
    struct Player *next;
}Player;

#endif
```

5.3 persistence.h

```
#ifndef PERSISTENCE_H
#define PERSISTENCE_H

#include "core_types.h"

/***
 * Saves the current state of the Player Hash Table to a binary file.
 */
void save_system_data();

/***
 * Loads player data from the binary file and reconstructs the Hash Table.
 */
void load_system_data();

/***
 * Generates a Management Report in CSV format.
 */
void export_daily_report(Machine machines[], int count);

#endif
```

5.4 main.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<time.h>
#include"core_types.h"
#include "persistence.h"

// Forward Declarations
void init_players();
void init_machines();
Player* create_player(const char* rfid, const char* name);
Player* find_player(const char* rfid);
void add_credits(Player* p, int amount);
int redeem_tickets(Player* p, int cost);
int transfer_credits(Player* sender, Player* receiver, int amount);
void free_players();
void update_surge_pricing();
int trigger_play_with_score(int machine_index, const char* player_name);
void run_watchdog_scan();
void print_ghost_logs(int machine_index);
void draw_dashboard();

extern Machine machines[];
extern int machine_count;

void flush_input() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}

int main() {
    srand(time(NULL));

    printf("[BOOT] Initializing Kernel...\n");
    init_players();
```

```
init_machines();
load_system_data();

if (find_player("TAG_ADMIN") == NULL) {
    create_player("TAG_ADMIN", "SysAdmin");
}

int running = 1;
char input_buffer[64];

while (running) {
    draw_dashboard();

    if (fgets(input_buffer, sizeof(input_buffer), stdin) == NULL)
break;
    char command = input_buffer[0];

    switch (command) {
        case 'q':
            running = 0;
            break;

        case 'n': // Register
            printf("Scan RFID Tag: ");
            char rfid[16];
            scanf("%15s", rfid);
            flush_input();

            Player *p =find_player(rfid);
            if (p) {
                printf("Welcome back, %s! Credits: %d\n", p->name,
p->credits);
            } else {
                printf("New Tag! Enter Name: ");
                char name[32];
                scanf("%31s", name);
                flush_input();
```

```

        p = create_player(rfid, name);
        add_credits(p, 20);
        printf("Registered!\n");
    }

    sleep(2);
    break;

case 'p': // Play with High Scores
    printf("Scan RFID to Play: ");
    char p_tag[16];
    scanf("%15s", p_tag);
    flush_input();

    Player *gamer = find_player(p_tag);
    if(gamer){
        printf("Identity: %s | Credits: %d\nEnter Machine ID:
", gamer->name, gamer->credits);
        int mid;
        scanf("%d", &mid);
        flush_input();

        if(gamer->credits > 0){
            gamer->credits--;
            trigger_play_with_score(mid, gamer->name);
            update_surge_pricing();
        } else {
            printf("Insufficient Credits!\n");
        }
    } else {
        printf("Unknown User.\n");
    }
    printf("Press Enter...");
```

getchar();

break;

case 't': // P2P Transfer

printf("--- P2P TRANSFER ---\n");

```

        printf("Sender RFID: ");
        char s_tag[16];
        scanf("%15s", s_tag);
        flush_input();

        Player *sender = find_player(s_tag);
        if(!sender) { printf("Sender not found.\n"); sleep(1);
break; }

        printf("Receiver RFID: ");
        char r_tag[16];
        scanf("%15s", r_tag);
        flush_input();

        Player *receiver = find_player(r_tag);
        if(!receiver) { printf("Receiver not found.\n"); sleep(1);
break; }

        printf("Amount to transfer: ");
        int amt;
        scanf("%d", &amt);
        flush_input();

        if(transfer_credits(sender, receiver, amt)){
            printf("SUCCESS! %s sent %d credits to %s.\n",
sender->name, amt, receiver->name);
        } else {
            printf("FAILED. Check balance.\n");
        }
        sleep(2);
        break;

    case 'w': // Watchdog
        run_watchdog_scan();
        printf("Press Enter...");
        getchar();
        break;

```

```

        case 'r': // Redeem
            printf("--- PRIZE STORE ---\n1. Eraser (10)\n2. Plushie
(1000)\nScan RFID: ");
            char shop_tag[16];
            scanf("%15s", shop_tag);
            flush_input();
            Player *shopper = find_player(shop_tag);
            if(shopper){
                printf("User: %s | Tix: %d\nItem ID: ",
shopper->name, shopper->tickets);
                int item_id, cost=0;
                scanf("%d", &item_id);
                flush_input();
                if(item_id == 1) cost = 10;
                else if(item_id == 2) cost = 1000;
                if(cost > 0) redeem_tickets(shopper, cost);
            }
            sleep(2);
            break;

        case 'l': // Logs
            printf("Machine ID: ");
            int lid;
            scanf("%d", &lid);
            flush_input();
            print_ghost_logs(lid);
            printf("Press Enter... ");
            getchar();
            break;

        case 'e': // Export
            export_daily_report(machines, machine_count);
            printf("Press Enter... ");
            getchar();
            break;
    }
}

```

```
}

printf("[SHUTDOWN] Saving system state...\n");
save_system_data();
free_players();
return 0;
}
```

5.5 machine_manager.c

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<time.h>
#include"core_types.h"

Machine machines[MAX_MACHINES];
int machine_count =0;

/***
 * Ghost Maintenance Logger.
 */
void log_machine_action(Machine *m, char type, int val){
    int idx = m->log_head;
    m->logs[idx].timestamp = time(NULL);
    m->logs[idx].action_type = type;
    m->logs[idx].value = val;
    m->log_head = ((m->log_head + 1) % GHOST_LOG_SIZE);
}

void init_machines(){
    for(int i=0; i<5; i++){
        machines[i].id = i;
        sprintf(machines[i].name, "Cabinet-%02d", i+1);
        machines[i].flags = STATUS_IDLE;
        machines[i].base_cost = 1.0;
        machines[i].cost_per_play = 1.0;
        machines[i].total_plays = 0;
        machines[i].session_plays = 0;
        machines[i].log_head = 0;
        machines[i].last_heartbeat = time(NULL);

        // Init Leaderboard
        machines[i].high_score = 500;
        strcpy(machines[i].champion_name, "CPU");
    }
}
```

```

        machine_count++;
    }

    printf("[SYS] Hardware Bus: %d units online.\n", machine_count);
}

/***
 * Dynamic Surge Pricing Algorithm.
 */
void update_surge_pricing(){
    for(int i=0; i<machine_count; i++) {
        if(machines[i].session_plays > 10) {
            machines[i].flags |= STATUS_HIGH_LOAD;
            machines[i].cost_per_play = machines[i].base_cost * 1.5;
        }
        else{
            machines[i].flags &= ~STATUS_HIGH_LOAD;
            machines[i].cost_per_play = machines[i].base_cost;
        }
    }
}

/***
 * Watchdog Timer
 * Scans for 'zombie' processes.
*/
void run_watchdog_scan(){
    time_t now = time(NULL);
    int fixed_count = 0;

    printf("[WDT] Scanning Hardware Bus...\n");

    for(int i=0; i<machine_count; i++) {
        Machine *m = &machines[i];

        if (m->flags & STATUS_PLAYING) {
            double elapsed = difftime(now, m->last_heartbeat);

```

```

// Threshold: 120 seconds
if(elapsed > 120.0){
    printf("[CRITICAL] Machine %d Watchdog Timeout! (Stuck for
%.0fs)\n", m->id, elapsed);

    m->flags = STATUS_IDLE;
    m->flags |= STATUS_JAMMED;

    log_machine_action(m, 'E', 999);
    fixed_count++;
}

}

if(fixed_count > 0){
    printf("[WDT] Watchdog recovered %d stuck units.\n", fixed_count);
} else {
    printf("[WDT] System Nominal.\n");
}
}

/***
 * Play Trigger with Scoring Logic
 */
int trigger_play_with_score(int machine_index, const char* player_name){
    if(machine_index < 0 || machine_index >= machine_count) return 0;

    Machine *m = &machines[machine_index];

    if(m->flags & STATUS_JAMMED){
        printf("[ERR] Machine %d is JAMMED. Technician required.\n",
m->id);
        return 0;
    }

    m->flags |= STATUS_PLAYING;
    m->total_plays++;
}

```

```

m->session_plays++;
m->last_heartbeat = time(NULL);

log_machine_action(m, 'P', 1);

// Gameplay Simulation
// Generate a random score between 100 and 1500
int score = (rand() % 1400) + 100;

if (score > m->high_score) {
    printf("\n*** NEW HIGH SCORE! ***\n");
    printf("%s beat %s with %d pts!\n", player_name, m->champion_name,
score);
    m->high_score = score;
    strncpy(m->champion_name, player_name, 31);
} else {
    printf("\nGame Over. Score: %d (High: %d by %s)\n", score,
m->high_score, m->champion_name);
}

return score;
}

void print_ghost_logs(int machine_index){
    if(machine_index < 0 || machine_index >= machine_count) return;
    Machine *m = &machines[machine_index];

    printf("\n--- GHOST LOGS: %s ---\n", m->name);
    for(int i=0; i<GHOST_LOG_SIZE; i++) {
        int idx = (m->log_head + i) % GHOST_LOG_SIZE;
        if(m->logs[idx].timestamp == 0) continue;

        printf("[%ld] Type: %c | Val: %d\n",
               m->logs[idx].timestamp, m->logs[idx].action_type,
m->logs[idx].value);
    }
    printf("-----\n");
}

```

```
}
```

5.6 persistence_manager.c

```
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include"core_types.h"

extern Player* create_player(const char* rfid, const char* name);
extern Player* player_table[MAX_PLAYERS_HASH];

typedef struct{
    char rfid_tag[ID_LEN];
    char name[32];
    int credits;
    int tickets;
    time_t last_seen;
}PlayerFileRecord;

void save_system_data() {
    FILE *fp = fopen("arcade_data.bin", "wb");
    if (!fp) {
        printf("[ERR] Disk Write Failure! Check permissions.\n");
        return;
    }

    int record_count = 0;
    for (int i = 0; i < MAX_PLAYERS_HASH; i++) {
        Player *current = player_table[i];
        while (current != NULL) {
            PlayerFileRecord record;
            memset(&record, 0, sizeof(PlayerFileRecord));

            strncpy(record.rfid_tag, current->rfid_tag, ID_LEN);
            strncpy(record.name, current->name, 32);
            record.credits = current->credits;
            record.tickets = current->tickets;
            record.last_seen = current->last_seen;
            if (fwrite(&record, sizeof(record), 1, fp) != 1) {
                printf("[ERR] Failed to write record %d to file.\n", record_count);
            }
            current = current->next;
        }
        record_count++;
    }
}
```

```

    record.last_seen = current->last_seen;

    fwrite(&record, sizeof(PlayerFileRecord), 1, fp);

    current = current->next;
    record_count++;
}
}

fclose(fp);
printf("[SYS] Persistence: %d records secured to
'arcade_data.bin'.\n", record_count);
}

void load_system_data() {
FILE *fp = fopen("arcade_data.bin", "rb");
if (!fp) {
    printf("[SYS] No backup found. Initializing fresh database.\n");
    return;
}

PlayerFileRecord record;
int count = 0;

while (fread(&record, sizeof(PlayerFileRecord), 1, fp) == 1) {
    Player *p = create_player(record.rfid_tag, record.name);
    if (p) {
        p->credits = record.credits;
        p->tickets = record.tickets;
        p->last_seen = record.last_seen;
        count++;
    }
}
fclose(fp);
printf("[SYS] Persistence: Restored %d players from disk.\n", count);
}

void export_daily_report(Machine machines[], int count) {

```

```
FILE *fp = fopen("daily_report.csv", "w");
if (!fp) {
    printf("[ERR] Could not write CSV report.\n");
    return;
}

fprintf(fp, "Machine ID,Name,Plays,Revenue,Status\n");
for(int i=0; i<count; i++){
    float revenue = machines[i].session_plays * machines[i].base_cost;
    char *status_txt = (machines[i].flags & STATUS_JAMMED) ? "JAMMED"
: "OK";

    fprintf(fp, "%d,%s,%d,%2f,%s\n",
            machines[i].id, machines[i].name, machines[i].session_plays,
revenue, status_txt
    );
}
fclose(fp);
printf("[SYS] Analytics: Data exported to 'daily_report.csv'.\n");
}
```

5.7 player_manager.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"core_types.h"

Player* player_table[MAX_PLAYERS_HASH];

unsigned long hash_func(const char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash % MAX_PLAYERS_HASH;
}

void init_players() {
    memset(player_table, 0, sizeof(player_table));
    printf("[SYS] Player Database Initialized (Capacity: %d)\n",
MAX_PLAYERS_HASH);
}

Player* find_player(const char* rfid) {
    unsigned long index = hash_func(rfid);
    Player *current = player_table[index];

    while (current != NULL) {
        if (strcmp(current->rfid_tag, rfid) == 0) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}

Player* create_player(const char* rfid, const char* name) {
    unsigned long index = hash_func(rfid);
```

```
Player *new_p = (Player*)malloc(sizeof(Player));
if (!new_p) {
    printf("[CRITICAL] Memory Allocation Failed!\n");
    return NULL;
}

strncpy(new_p->rfid_tag, rfid, ID_LEN);
strncpy(new_p->name, name, 31);
new_p->credits = 0;
new_p->tickets = 0;
new_p->last_seen = time(NULL);

new_p->next = player_table[index];
player_table[index] = new_p;

return new_p;
}

void add_credits(Player* p, int amount){
    if(p) p->credits += amount;
}

/**
 * Peer-to-Peer Transfer
 */
int transfer_credits(Player* sender, Player* receiver, int amount){
    if (!sender || !receiver) return 0;

    if (sender->credits >= amount) {
        sender->credits -= amount;
        receiver->credits += amount;
        return 1;
    }
    return 0;
}
```

```
/**  
 * Transactional Redemption  
 */  
  
int redeem_tickets(Player* p, int cost){  
    if (!p) return 0;  
  
    if (p->tickets >= cost){  
        p->tickets -= cost;  
        printf("[TXN] Redemption Success! Items dispensed. Balance: %d\n",  
p->tickets);  
        return 1;  
    } else {  
        printf("[TXN] Insufficient Tickets. Need: %d, Have: %d\n", cost,  
p->tickets);  
        return 0;  
    }  
}  
  
void free_players(){  
    for (int i = 0; i < MAX_PLAYERS_HASH; i++){  
        Player *current = player_table[i];  
        while (current != NULL) {  
            Player *temp = current;  
            current = current->next;  
            free(temp);  
        }  
    }  
}
```

5.8 ui_renderer.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "core_types.h"

extern Machine machines[];
extern int machine_count;

#define RED    "\x1B[31m"
#define GRN    "\x1B[32m"
#define YEL    "\x1B[33m"
#define RESET  "\x1B[0m"
#define CLR    "\033[2J\033[1;1H"

void draw_dashboard() {
    printf("%s", CLR);

    printf("=====\\n");
    printf(" ARCADE KERNEL v4.2 [COMPETITIVE] | Time: %ld\\n", time(NULL));

    printf("=====\\n");
    printf("| %-10s | %-8s | %-5s | %-5s | %-5s | %-12s | \\n", "ID",
"STATUS", "COST", "PLAYS", "BEST", "CHAMPION");

    printf("| ----- | ----- | ----- | ----- | ----- | ----- | ----- | \\n");
}

for (int i = 0; i < machine_count; i++) {
    Machine *m = &machines[i];

    char status_str[16];
    char *color = RESET;

    if (m->flags & STATUS_JAMMED) {
```

```
    strcpy(status_str, "JAMMED");
    color = RED;
}

else if (m->flags & STATUS_PLAYING) {
    strcpy(status_str, "PLAYING");
    color = GRN;
}

else{
    strcpy(status_str, "IDLE");
}

printf(" | %-10s | %s%-8s%s | %5.2f | %-5d | %-5d | %-12s |\n",
       m->name, color, status_str, RESET,
       m->cost_per_play, m->session_plays,
       m->high_score, m->champion_name);
}

printf("=====\\n");
printf("USER: [n] Register [p] Play [r] Redeem [t] Transfer\\n");
printf("SYS:  [l] Logs      [e] Export [w] Watchdog [q] Quit\\n");
printf("> ");
}
```

6. APPENDIX B: OUTPUT

```
ARCADE KERNEL v4.2 [COMPETITIVE] | Time: 1764826154
=====
| ID   | STATUS | COST | PLAYS | BEST | CHAMPION |
|-----|
| Cabinet-01 | IDLE | 1.00 | 0 | 500 | CPU |
| Cabinet-02 | PLAYING | 1.00 | 1 | 840 | Raghav |
| Cabinet-03 | IDLE | 1.00 | 0 | 500 | CPU |
| Cabinet-04 | IDLE | 1.00 | 0 | 500 | CPU |
| Cabinet-05 | IDLE | 1.00 | 0 | 500 | CPU |

USER: [n] Register [p] Play [r] Redeem [t] Transfer
SYS: [l] Logs [e] Export [w] Watchdog [q] Quit
> p
Scan RFID to Play: 1234
Identity: Raghav | Credits: 19
Enter Machine ID: 1

Game Over. Score: 651 (High: 840 by Raghav)
Press Enter...[
```



```
ARCADE KERNEL v4.2 [COMPETITIVE] | Time: 1764826227
=====
| ID   | STATUS | COST | PLAYS | BEST | CHAMPION |
|-----|
| Cabinet-01 | IDLE | 1.00 | 0 | 500 | CPU |
| Cabinet-02 | PLAYING | 1.00 | 2 | 840 | Raghav |
| Cabinet-03 | IDLE | 1.00 | 0 | 500 | CPU |
| Cabinet-04 | IDLE | 1.00 | 0 | 500 | CPU |
| Cabinet-05 | IDLE | 1.00 | 0 | 500 | CPU |

USER: [n] Register [p] Play [r] Redeem [t] Transfer
SYS: [l] Logs [e] Export [w] Watchdog [q] Quit
> r
--- PRIZE STORE ---
1. Eraser (10)
2. Plushie (1000)
Scan RFID: 1234
User: Raghav | Tix: 0
Item ID: 1
[TXN] Insufficient Tickets. Need: 10, Have: 0
```

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL PORTS

=====
ARCADE KERNEL v4.2 [COMPETITIVE] | Time: 1764826243
=====

ID	STATUS	COST	PLAYS	BEST	CHAMPION
Cabinet-01	IDLE	1.00	0	500	CPU
Cabinet-02	PLAYING	1.00	2	840	Raghav
Cabinet-03	IDLE	1.00	0	500	CPU
Cabinet-04	IDLE	1.00	0	500	CPU
Cabinet-05	IDLE	1.00	0	500	CPU

USER: [n] Register [p] Play [r] Redeem [t] Transfer

SYS: [l] Logs [e] Export [w] Watchdog [q] Quit

> l

Machine ID: 1

--- GHOST LOGS: Cabinet-02 ---

[1764826148] Type: P | Val: 1

[1764826163] Type: P | Val: 1

Press Enter...█

=====
ARCADE KERNEL v4.2 [COMPETITIVE] | Time: 1764826325
=====

ID	STATUS	COST	PLAYS	BEST	CHAMPION
Cabinet-01	IDLE	1.00	0	500	CPU
Cabinet-02	IDLE	1.00	0	500	CPU
Cabinet-03	IDLE	1.00	0	500	CPU
Cabinet-04	IDLE	1.00	0	500	CPU
Cabinet-05	IDLE	1.00	0	500	CPU

USER: [n] Register [p] Play [r] Redeem [t] Transfer

SYS: [l] Logs [e] Export [w] Watchdog [q] Quit

> e

[SYS] Analytics: Data exported to 'daily_report.csv'.

Press Enter...█

The screenshot shows a terminal window with several tabs at the top. The active tab is 'daily_report.csv'. The content of the terminal is as follows:

```
core_types.h machine_manager.c player_manager.c main.c daily_report.csv ui_renderer.c pers >
daily_report.csv > data
1 Machine ID,Name,Plays,Revenue,Status
2 0,Cabinet-01,0,0.00,OK
3 1,Cabinet-02,0,0.00,OK
4 2,Cabinet-03,0,0.00,OK
5 3,Cabinet-04,0,0.00,OK
6 4,Cabinet-05,0,0.00,OK
7
```