```
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/poll.h>
#include <unistd.h>
#include <time.h>
#include <termios.h>

#include "rk_debug.h"
#include "rk_defines.h"
#include "rk_mpi_adec.h"
#include "rk_mpi_aenc.h"
#include "rk_mpi_ai.h"
#include "rk_mpi_ao.h"
#include "rk_mpi_avs.h"
#include "rk_mpi_cal.h"
#include "rk_mpi_ivs.h"
#include "rk_mpi_mb.h"
#include "rk_mpi_rgn.h"
#include "rk_mpi_sys.h"
#include "rk_mpi_tde.h"
#include "rk_mpi_vdec.h"
#include "rk_mpi_venc.h"
#include "rk_mpi_vi.h"
#include "rk_mpi_vo.h"
#include "rk_mpi_vpss.h"
#include "linux/input.h"
#include <fcntl.h>

#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <linux/i2c-dev.h>
#include <linux/rtc.h>

#define MAX_PATH_LEN 512  // bigger and safer

static RGN_HANDLE g_rgn_handle = 1;
static bool quit = false;
FILE *file = NULL;
static RK_CHAR *g_pOutPath = "/mnt/sdcard/";
char folder_path[MAX_PATH_LEN];
int B = 1;
static bool is_night_mode = false;

static pthread_mutex_t g_rgn_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
static RK_CHAR optstr[] = "?::w:h:I:e:";

// GPS related variables - initialized once and reused
static bool g_gps_initialized = false;
static bool g_gps_coordinates_obtained = false;
static char g_gps_latitude[32] = "N/A";
static char g_gps_longitude[32] = "N/A";
static pthread_mutex_t g_gps_mutex = PTHREAD_MUTEX_INITIALIZER;
int bcd_to_dec(unsigned char val);
int getTimeRTC(void);
unsigned char dec_to_bcd(int val);
int setRTCAlarm(int alarm_hour, int add_minutes);
static void make_dir_if_not_exists(const char *path);
void create_full_date_folder(char *folder_path);
void update_rgn_timestamp();
int x=1;

static bool capture_triggered = false;
static pthread_mutex_t capture_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t capture_cond = PTHREAD_COND_INITIALIZER;

// Function to set serial port baud rate
bool set_serial_baudrate(int fd, int baudrate) {
    struct termios options;

    // Get current serial port settings
    if (tcgetattr(fd, &options) != 0) {
        perror("tcgetattr failed");
        return false;
    }

    // Set baud rate
    cfsetispeed(&options, B9600);
    cfsetospeed(&options, B9600);

    // Configure other serial port settings
    options.c_cflag &= ~PARENB;   // No parity
    options.c_cflag &= ~CSTOPB;   // 1 stop bit
    options.c_cflag &= ~CSIZE;    // Clear data bit mask
    options.c_cflag |= CS8;       // 8 data bits
    options.c_cflag &= ~CRTSCTS;  // No hardware flow control
    options.c_cflag |= (CLOCAL | CREAD);  // Enable receiver, ignore modem control
lines

    // Configure input modes
    options.c_iflag &= ~(IXON | IXOFF | IXANY);  // No software flow control
    options.c_iflag &= ~(INLCR | ICRNL);         // No input processing

    // Configure output modes
```

```
    options.c_oflag &= ~OPOST;  // Raw output

    // Configure local modes
    options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);  // Raw input

    // Set the new attributes
    if (tcsetattr(fd, TCSANOW, &options) != 0) {
        perror("tcsetattr failed");
        return false;
    }

    printf("Serial port configured: 9600 baud, 8N1, no flow control\n");
    return true;
}

static void print_usage(const RK_CHAR *name) {
    printf("usage example:\n");
    printf("\t%s -I 0 -w 1920 -h 1080 \n", name);
    printf("\t-w | --width: VI width, Default:1920\n");
    printf("\t-h | --heght: VI height, Default:1080\n");
    printf("\t-I | --camid: camera ctx id, Default 0. "
           "0:rkisp_mainpath,1:rkisp_selfpath,2:rkisp_bypasspath\n");
}

static void sigterm_handler(int sig) {
    fprintf(stderr, "signal %d\n", sig);
    quit = true;
}

// Check LDR and set day/night mode
void check_ldr_and_set_mode(void) {
    int ldr_fd = open("/sys/class/gpio/gpio72/value", O_RDONLY);
    if (ldr_fd >= 0) {
        char val;
        if (read(ldr_fd, &val, 1) > 0) {
            if (val == '0') {
                is_night_mode = false; // DAY mode
                printf("LDR: DAY mode - Using BLACK text\n");
            } else {
                is_night_mode = true; // NIGHT mode
                printf("LDR: NIGHT mode - Using WHITE text\n");
            }
        }
        close(ldr_fd);
    } else {
        is_night_mode = true;
        printf("LDR read failed - Defaulting to NIGHT mode\n");
    }
}
```

```
// Parse GPS Data from RMC format - FIXED VERSION
// Parse GPS Data from RMC format - Convert to Decimal Degrees
void parse_gps_data(const char *gps_data) {
    pthread_mutex_lock(&g_gps_mutex);

    printf("1RAW GPS DATA: %s\n", gps_data);

    char data_copy[512];
    strncpy(data_copy, gps_data, sizeof(data_copy) - 1);
    data_copy[sizeof(data_copy) - 1] = '\0';

    // Extract just the NMEA sentence from the response
    char *nmea_start = strstr(data_copy, "$GNRMC");
    if (!nmea_start) {
        nmea_start = strstr(data_copy, "$GPRMC");
    }

    if (!nmea_start) {
        printf("No NMEA sentence found in response\n");
        pthread_mutex_unlock(&g_gps_mutex);
        return;
    }

    // Copy only the NMEA sentence for parsing
    char nmea_sentence[256];
    char *nmea_end = strchr(nmea_start, '*');
    if (nmea_end) {
        // Include the checksum
        nmea_end += 3; // * + 2 checksum chars
        size_t nmea_len = nmea_end - nmea_start;
        if (nmea_len < sizeof(nmea_sentence)) {
            strncpy(nmea_sentence, nmea_start, nmea_len);
            nmea_sentence[nmea_len] = '\0';
        } else {
            strncpy(nmea_sentence, nmea_start, sizeof(nmea_sentence) - 1);
            nmea_sentence[sizeof(nmea_sentence) - 1] = '\0';
        }
    } else {
        // No checksum found, just copy to next line end
        strncpy(nmea_sentence, nmea_start, sizeof(nmea_sentence) - 1);
        nmea_sentence[sizeof(nmea_sentence) - 1] = '\0';
        char *line_end = strchr(nmea_sentence, '\r');
        if (line_end) *line_end = '\0';
        line_end = strchr(nmea_sentence, '\n');
        if (line_end) *line_end = '\0';
    }

    printf("Extracted NMEA: %s\n", nmea_sentence);

    // Now parse the clean NMEA sentence
```

```
    char *token = strtok(nmea_sentence, ",");
    int field = 0;
    char latitude[32] = "N/A";
    char longitude[32] = "N/A";
    char lat_dir[2] = "";
    char lon_dir[2] = "";
    char status[2] = "";

    while (token != NULL) {
        field++;

        // For RMC format
        if (field == 2 && strlen(token) > 0) { // UTC Time
            printf("UTC Time: %s\n", token);
        }
        else if (field == 3 && strlen(token) > 0) { // Status A=active,V=void
            strncpy(status, token, sizeof(status) - 1);
            printf("Status: %s\n", status);
        }
        else if (field == 4 && strlen(token) > 0) { // Latitude
            strncpy(latitude, token, sizeof(latitude) - 1);
        }
        else if (field == 5 && strlen(token) > 0) { // Latitude direction
            strncpy(lat_dir, token, sizeof(lat_dir) - 1);
        }
        else if (field == 6 && strlen(token) > 0) { // Longitude
            strncpy(longitude, token, sizeof(longitude) - 1);
        }
        else if (field == 7 && strlen(token) > 0) { // Longitude direction
            strncpy(lon_dir, token, sizeof(lon_dir) - 1);
        }

        token = strtok(NULL, ",");
    }

    printf("Parsed - Status: %s, Lat: %s %s, Lon: %s %s\n", status, latitude,
lat_dir, longitude, lon_dir);

    // Only use coordinates if status is 'A' (active) and coordinates are valid
    if (status[0] == 'A' && strlen(latitude) > 0 && strlen(longitude) > 0 &&
        strlen(latitude) < 20 && strlen(longitude) < 20) {

        // Convert NMEA format (DDMM.MMMMM) to decimal degrees (DD.DDDDDDD)
        double lat_degrees, lon_degrees;

        // Convert latitude: format is DDMM.MMMMM
        double lat_nmea = atof(latitude);
        int lat_deg = (int)(lat_nmea / 100);  // Get degrees part
        double lat_min = lat_nmea - (lat_deg * 100);  // Get minutes part
        lat_degrees = lat_deg + (lat_min / 60.0);  // Convert to decimal degrees
```

```
        if (lat_dir[0] == 'S') lat_degrees = -lat_degrees;  // Southern hemisphere

        // Convert longitude: format is DDDMM.MMMMM
        double lon_nmea = atof(longitude);
        int lon_deg = (int)(lon_nmea / 100);  // Get degrees part
        double lon_min = lon_nmea - (lon_deg * 100);  // Get minutes part
        lon_degrees = lon_deg + (lon_min / 60.0);  // Convert to decimal degrees
        if (lon_dir[0] == 'W') lon_degrees = -lon_degrees;  // Western hemisphere

        // Format with 7 decimal places (high precision)
        snprintf(g_gps_latitude, sizeof(g_gps_latitude), "%.7f", lat_degrees);
        snprintf(g_gps_longitude, sizeof(g_gps_longitude), "%.7f", lon_degrees);

        printf("CONVERTED GPS: Lat=%s, Lon=%s\n", g_gps_latitude, g_gps_longitude);
           // Validate the coordinates make sense
        double lat_val = atof(g_gps_latitude);
        double lon_val = atof(g_gps_longitude);

        if (lat_val < -90 || lat_val > 90) {
        printf("ERROR: Invalid latitude: %f\n", lat_val);
        snprintf(g_gps_latitude, sizeof(g_gps_latitude), "N/A");
        }
        if (lon_val < -180 || lon_val > 180) {
         printf("ERROR: Invalid longitude: %f\n", lon_val);
        snprintf(g_gps_longitude, sizeof(g_gps_longitude), "N/A");
        }

    } else {
        printf("INVALID GPS DATA - Status: %s, or wrong format\n", status);
        snprintf(g_gps_latitude, sizeof(g_gps_latitude), "N/A");
        snprintf(g_gps_longitude, sizeof(g_gps_longitude), "N/A");
    }

    pthread_mutex_unlock(&g_gps_mutex);
}
// Get GPS Coordinates Function - Called only once during initialization
bool get_gps_coordinates_once(void) {
    if (!g_gps_initialized) {
        printf("GPS not initialized\n");
        return false;
    }

    printf("Attempting to get GPS coordinates...\n");

    // Open GPS device with proper settings
    int gps_fd = open("/dev/ttyUSB6", O_RDWR | O_NOCTTY | O_NONBLOCK);
    if (gps_fd < 0) {
        printf("ERROR: Cannot open GPS device for writing\n");
        return false;
    }
```

```
    // Set baud rate to 9600
    if (!set_serial_baudrate(gps_fd, 9600)) {
        printf("ERROR: Failed to set baud rate\n");
        close(gps_fd);
        return false;
    }

    // Request RMC NMEA data
    char gps_cmd[] = "AT+QGPSGNMEA=\"RMC\"\r\n";
    write(gps_fd, gps_cmd, strlen(gps_cmd));
    printf("Sent: AT+QGPSGNMEA=\"RMC\"\n");

    // Give time for GPS to respond
    sleep(5);

    // Read the response
    char response[512];
    int attempts = 0;
    ssize_t bytes_read;

    // Try multiple times to read GPS data
    while (attempts < 10) {
        bytes_read = read(gps_fd, response, sizeof(response) - 1);

        if (bytes_read > 0) {
            response[bytes_read] = '\0';
            printf("GPS Raw Response: %s\n", response);

            // Check if we have valid GPS data (not void)
            if (strstr(response, "V,,,,,,,") == NULL &&
                (strstr(response, "GNRMC") != NULL || strstr(response, "GPRMC") !=
NULL)) {
                parse_gps_data(response);
                close(gps_fd);
                g_gps_coordinates_obtained = true;
                return true;
            } else {
                printf("GPS data is void or no RMC data, retrying...\n");
            }
        } else {
            printf("No data received, retrying...\n");
        }

        attempts++;
        sleep(2);

        // Resend command if needed
        if (attempts % 3 == 0) {
            write(gps_fd, gps_cmd, strlen(gps_cmd));
```

```
            printf("Resent GPS command\n");
        }
    }


    close(gps_fd);
    printf("Failed to get valid GPS coordinates after %d attempts\n", attempts);
    return false;
}

// GPS Initialization Function - Called only once
bool initialize_gps_module(void) {
    printf("=== GPS INITIALIZATION (ONE-TIME) ===\n");

    // Load option module for GSM modem
    system("modprobe option 2>/dev/null");
    printf("Loaded option module\n");

    // Register the device ID for the GSM modem
    system("echo \"2c7c 0901\" > /sys/bus/usb-serial/drivers/option1/new_id
2>/dev/null");
    printf("Registered device ID 2c7c:0901\n");

    // Wait for devices to be created
    sleep(3);

    // Try to initialize GPS on ttyUSB6 (based on your microcom command)
    int gps_fd = open("/dev/ttyUSB6", O_RDWR | O_NOCTTY | O_NONBLOCK);
    if (gps_fd < 0) {
        printf("ERROR: Cannot open /dev/ttyUSB6 for GPS\n");

        // Try alternative devices including ttyUSB6
        for (int i = 0; i <= 6; i++) {
            char device[32];
            snprintf(device, sizeof(device), "/dev/ttyUSB%d", i);
            gps_fd = open(device, O_RDWR | O_NOCTTY | O_NONBLOCK);
            if (gps_fd >= 0) {
                printf("Found GPS device at %s\n", device);
                break;
            }
        }

        if (gps_fd < 0) {
            printf("ERROR: No GPS device found\n");
            return false;
        }
    }

    printf("GPS device opened successfully\n");
```

```
    // Set baud rate to 9600
    if (!set_serial_baudrate(gps_fd, 9600)) {
        printf("ERROR: Failed to set baud rate to 9600\n");
        close(gps_fd);
        return false;
    }

    // Send GPS initialization commands
    char init_cmd1[] = "AT+QGPS=1\r\n";
    char init_cmd2[] = "AT+QGPS?\r\n";

    write(gps_fd, init_cmd1, strlen(init_cmd1));
    printf("Sent: AT+QGPS=1\n");
    sleep(3);

    write(gps_fd, init_cmd2, strlen(init_cmd2));
    printf("Sent: AT+QGPS?\n");
    sleep(2);

    close(gps_fd);

    printf("GPS module initialized successfully at 9600 baud\n");
    g_gps_initialized = true;

    // Get coordinates once and store them
    if (get_gps_coordinates_once()) {
        printf("=== GPS COORDINATES OBTAINED SUCCESSFULLY ===\n");
        printf("Latitude: %s\n", g_gps_latitude);
        printf("Longitude: %s\n", g_gps_longitude);
        printf("=== THESE COORDINATES WILL BE USED FOR ALL IMAGES ===\n");
    } else {
        printf("WARNING: Could not obtain GPS coordinates\n");
    }
    system("echo 1 > /sys/class/gpio/gpio53/value");
    usleep(10000);
    system("echo 0 > /sys/class/gpio/gpio53/value");
    return true;
}

// Enhanced text drawing function with full character support
void draw_large_text(uint32_t *buffer, int buf_width, int x, int y, const char
*text, uint32_t color) {
    // Enhanced 5x7 font for ALL characters including letters and decimal point
    static const uint8_t enhanced_font[256][7] = {
        // Numbers 0-9
        [48] = {0x3E, 0x7F, 0xFF, 0xFF, 0xFF, 0x7F, 0x3E}, // 0
        [49] = {0x1C, 0x3C, 0x1C, 0x1C, 0x1C, 0x1C, 0x7F}, // 1
        [50] = {0x3E, 0x7F, 0x07, 0x3E, 0x70, 0x7F, 0x3E}, // 2
        [51] = {0x3E, 0x7F, 0x07, 0x1E, 0x07, 0x7F, 0x3E}, // 3
        [52] = {0x67, 0x67, 0x67, 0x7F, 0x3F, 0x07, 0x07}, // 4
```

```
[53] = {0x3F, 0x7F, 0x70, 0x7E, 0x07, 0x7F, 0x3E}, // 5
[54] = {0x3E, 0x7F, 0x70, 0x7E, 0x67, 0x7F, 0x3E}, // 6
[55] = {0x3F, 0x7F, 0x07, 0x0E, 0x1C, 0x38, 0x38}, // 7
[56] = {0x3E, 0x7F, 0x67, 0x3E, 0x67, 0x7F, 0x3E}, // 8
[57] = {0x3E, 0x7F, 0x67, 0x3F, 0x07, 0x7F, 0x3E}, // 9

// Letters A-Z, a-z
[65] = {0x1C, 0x3E, 0x77, 0x63, 0x7F, 0x7F, 0x63}, // A
[66] = {0x7E, 0x7F, 0x63, 0x7E, 0x63, 0x7F, 0x7E}, // B
[67] = {0x3E, 0x7F, 0x61, 0x60, 0x61, 0x7F, 0x3E}, // C
[68] = {0x7E, 0x7F, 0x63, 0x63, 0x63, 0x7F, 0x7E}, // D
[69] = {0x7F, 0x7F, 0x60, 0x7E, 0x60, 0x7F, 0x7F}, // E
[70] = {0x7F, 0x7F, 0x60, 0x7E, 0x60, 0x60, 0x60}, // F
[71] = {0x3E, 0x7F, 0x61, 0x67, 0x63, 0x7F, 0x3F}, // G
[72] = {0x63, 0x63, 0x63, 0x7F, 0x7F, 0x63, 0x63}, // H
[73] = {0x3E, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x3E}, // I
[74] = {0x0F, 0x07, 0x07, 0x07, 0x67, 0x7F, 0x3E}, // J
[75] = {0x63, 0x66, 0x6C, 0x78, 0x6C, 0x66, 0x63}, // K
[76] = {0x60, 0x60, 0x60, 0x60, 0x60, 0x7F, 0x7F}, // L
[77] = {0x63, 0x77, 0x7F, 0x6B, 0x63, 0x63, 0x63}, // M
[78] = {0x63, 0x73, 0x7B, 0x6F, 0x67, 0x63, 0x63}, // N
[79] = {0x3E, 0x7F, 0x63, 0x63, 0x63, 0x7F, 0x3E}, // O
[80] = {0x7E, 0x7F, 0x63, 0x7F, 0x7E, 0x60, 0x60}, // P
[81] = {0x3E, 0x7F, 0x63, 0x63, 0x6B, 0x7F, 0x3F}, // Q
[82] = {0x7E, 0x7F, 0x63, 0x7F, 0x7E, 0x66, 0x63}, // R
[83] = {0x3E, 0x7F, 0x60, 0x3E, 0x03, 0x7F, 0x3E}, // S
[84] = {0x7F, 0x7F, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C}, // T
[85] = {0x63, 0x63, 0x63, 0x63, 0x63, 0x7F, 0x3E}, // U
[86] = {0x63, 0x63, 0x63, 0x77, 0x3E, 0x1C, 0x08}, // V
[87] = {0x63, 0x63, 0x63, 0x6B, 0x7F, 0x77, 0x63}, // W
[88] = {0x63, 0x77, 0x3E, 0x1C, 0x3E, 0x77, 0x63}, // X
[89] = {0x63, 0x77, 0x3E, 0x1C, 0x1C, 0x1C, 0x1C}, // Y
[90] = {0x7F, 0x7F, 0x0E, 0x1C, 0x38, 0x7F, 0x7F}, // Z

// Lowercase letters
[97] = {0x00, 0x3E, 0x03, 0x3F, 0x67, 0x67, 0x3F}, // a
[98] = {0x60, 0x60, 0x7E, 0x7F, 0x63, 0x7F, 0x7E}, // b
[99] = {0x00, 0x3E, 0x7F, 0x60, 0x60, 0x7F, 0x3E}, // c
[100] = {0x03, 0x03, 0x3F, 0x7F, 0x63, 0x7F, 0x3F}, // d
[101] = {0x00, 0x3E, 0x7F, 0x7F, 0x7E, 0x7F, 0x3E}, // e
[102] = {0x0F, 0x1F, 0x18, 0x3E, 0x3E, 0x18, 0x18}, // f
[103] = {0x00, 0x3F, 0x7F, 0x63, 0x3F, 0x03, 0x7E}, // g
[104] = {0x60, 0x60, 0x7E, 0x7F, 0x63, 0x63, 0x63}, // h
[105] = {0x18, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18}, // i
[106] = {0x06, 0x00, 0x06, 0x06, 0x06, 0x66, 0x3C}, // j
[107] = {0x60, 0x66, 0x6C, 0x78, 0x6C, 0x66, 0x63}, // k
[108] = {0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18}, // l
[109] = {0x00, 0x77, 0x7F, 0x6B, 0x6B, 0x63, 0x63}, // m
[110] = {0x00, 0x7E, 0x7F, 0x63, 0x63, 0x63, 0x63}, // n
[111] = {0x00, 0x3E, 0x7F, 0x63, 0x63, 0x7F, 0x3E}, // o
```

```
        // Punctuation and symbols
        [46] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x18}, // . (decimal point)
        [58] = {0x00, 0x1C, 0x1C, 0x00, 0x1C, 0x1C, 0x00}, // :
        [45] = {0x00, 0x00, 0x00, 0x7F, 0x7F, 0x00, 0x00}, // -
        [32] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}  // space
    };

    int char_width = 20;
    uint32_t border_color = 0xFF000000;

    for (int i = 0; text[i] != '\0'; i++) {
        int char_start_x = x + i * (char_width + 2);
        int ascii_code = (int)text[i];

        // Use enhanced font or default to space if character not defined
        const uint8_t *char_data = enhanced_font[ascii_code];
        if (char_data[0] == 0 && char_data[1] == 0 && char_data[2] == 0) {
            // Character not defined, use space
            char_data = enhanced_font[32];
        }

        // Draw character
        for (int row = 0; row < 7; row++) {
            uint8_t row_data = char_data[row];
            for (int col = 0; col < 8; col++) {
                if (row_data & (1 << (7 - col))) {
                    for (int py = 0; py < 4; py++) {
                        for (int px = 0; px < 3; px++) {
                            int pixel_x = char_start_x + col * 3 + px;
                            int pixel_y = y + row * 4 + py;
                            if (pixel_x < buf_width && pixel_y < 80) {
                                if (px == 0 || px == 2 || py == 0 || py == 3) {
                                    buffer[pixel_y * buf_width + pixel_x] =
border_color;
                                } else {
                                    buffer[pixel_y * buf_width + pixel_x] = color;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

// Updated RGN initialization with better configuration
void init_timestamp_rgn(int width, int height) {
    RGN_ATTR_S stAttr;
```

```
    RGN_CHN_ATTR_S stChnAttr;
    MPP_CHN_S stChn;

    printf("Initializing timestamp overlay for resolution: %dx%d\n", width, height);

    memset(&stAttr, 0, sizeof(RGN_ATTR_S));
    stAttr.enType = OVERLAY_RGN;
    stAttr.unAttr.stOverlay.enPixelFmt = RK_FMT_ARGB8888;
    stAttr.unAttr.stOverlay.stSize.u32Width = 1100;
    stAttr.unAttr.stOverlay.stSize.u32Height = 80; // Increased height for GPS

    RK_S32 ret = RK_MPI_RGN_Create(g_rgn_handle, &stAttr);
    if (ret != RK_SUCCESS) {
        RK_LOGE("Failed to create RGN: 0x%x", ret);
        return;
    }
    printf("RGN created successfully - Size: 1050x80\n");

    memset(&stChnAttr, 0, sizeof(RGN_CHN_ATTR_S));
    stChnAttr.bShow = RK_TRUE;
    stChnAttr.enType = OVERLAY_RGN;

    // Position at bottom center
    int overlay_x = (width - 1100) / 2;
    int overlay_y = height - 100;

    stChnAttr.unChnAttr.stOverlayChn.stPoint.s32X = overlay_x;
    stChnAttr.unChnAttr.stOverlayChn.stPoint.s32Y = overlay_y;
    stChnAttr.unChnAttr.stOverlayChn.u32FgAlpha = 255;
    stChnAttr.unChnAttr.stOverlayChn.u32BgAlpha = 220; // Higher opacity for better
visibility
    stChnAttr.unChnAttr.stOverlayChn.u32Layer = 1;

    stChn.enModId = RK_ID_VENC;
    stChn.s32DevId = 0;
    stChn.s32ChnId = 0;

    ret = RK_MPI_RGN_AttachToChn(g_rgn_handle, &stChn, &stChnAttr);
    if (ret != RK_SUCCESS) {
        RK_LOGE("Failed to attach RGN to channel: 0x%x", ret);
        RK_MPI_RGN_Destroy(g_rgn_handle);
        return;
    }
    printf("RGN attached to VENC channel successfully at (%d, %d)\n", overlay_x,
overlay_y);

    // Check initial LDR state
    check_ldr_and_set_mode();
    update_rgn_timestamp();
}
```

```
// Also update the text drawing to use more horizontal space
void update_rgn_timestamp() {
    pthread_mutex_lock(&g_rgn_mutex);

    RGN_CANVAS_INFO_S stCanvasInfo;
    char display_text[256];  // Increased buffer for one line
    time_t now = time(NULL);
    struct tm *tm = localtime(&now);

    if (RK_MPI_RGN_GetCanvasInfo(g_rgn_handle, &stCanvasInfo) != RK_SUCCESS) {
        printf("ERROR: Failed to get canvas info\n");
        pthread_mutex_unlock(&g_rgn_mutex);
        return;
    }

    // Format everything in one line: Date Time Lat Lon
    snprintf(display_text, sizeof(display_text),
             "%04d-%02d-%02d %02d:%02d:%02d lat:%s lon:%s",
             tm->tm_year + 1900, tm->tm_mon + 1, tm->tm_mday,
             tm->tm_hour, tm->tm_min, tm->tm_sec,
             g_gps_latitude, g_gps_longitude);

    printf("Display Text: %s\n", display_text);
    printf("Canvas Size: %dx%d\n", stCanvasInfo.u32VirWidth,
stCanvasInfo.u32VirHeight);

    uint32_t *pData = (uint32_t *)(uintptr_t)stCanvasInfo.u64VirAddr;

    if (!pData) {
        printf("ERROR: Canvas virtual address is NULL\n");
        pthread_mutex_unlock(&g_rgn_mutex);
        return;
    }

    // Choose colors based on day/night mode
    uint32_t text_color, bg_color;

    //if (is_night_mode) {
     //   text_color = 0xFFFFFFFF; // White text
      //  bg_color = 0xDD000000;   // Dark semi-transparent background
    //} else {
        text_color = 0xFF000000; // Black text
        bg_color = 0xDDFFFFFF;   // Light semi-transparent background
    //}

    // Clear entire canvas with background color
    for (int i = 0; i < stCanvasInfo.u32VirWidth * stCanvasInfo.u32VirHeight; i++) {
        pData[i] = bg_color;
```

```
    }

    // Draw single line of text - start from left edge with some padding
    int text_x = 10;  // Reduced from 20 to start more left
    int text_y = 35;  // Centered vertically in the 80px height

    draw_large_text(pData, stCanvasInfo.u32VirWidth, text_x, text_y, display_text,
text_color);

    if (RK_MPI_RGN_UpdateCanvas(g_rgn_handle) != RK_SUCCESS) {
        printf("ERROR: Failed to update canvas\n");
    }

    pthread_mutex_unlock(&g_rgn_mutex);
}

static void *GetMediaBuffer0(void *arg) {
    (void)arg;
    printf("========%s========\n", __func__);
    void *pData = RK_NULL;
    int s32Ret;
    static RK_U32 jpeg_id = 0;
    char jpeg_path[MAX_PATH_LEN];
    char thumb_path[MAX_PATH_LEN];

    FILE *file = NULL;
    FILE *thumb_file = NULL;

    VENC_STREAM_S stFrameMain;
    VENC_STREAM_S stFrameThumb;

    stFrameMain.pstPack = calloc(1, sizeof(VENC_PACK_S));
    stFrameThumb.pstPack = calloc(1, sizeof(VENC_PACK_S));

    if (!stFrameMain.pstPack || !stFrameThumb.pstPack) {
        RK_LOGE("Failed to allocate memory for VENC_PACK_S");
        return NULL;
    }

    while (!quit) {

        // 🆕 STEP 1: Create year/month/day folder automatically
        create_full_date_folder(folder_path);

        // 🆕 STEP 2: Get current time for timestamped filename
        time_t now = time(NULL);
        struct tm *t = localtime(&now);

        // 🆕 STEP 3: Build full image path (inside date folder)
        memset(jpeg_path, 0, sizeof(jpeg_path));
```

```
        memset(thumb_path, 0, sizeof(thumb_path));

        snprintf(jpeg_path, sizeof(jpeg_path),
                 "%s/IMG_%04d%02d%02d_%02d%02d%02d.jpeg",
                 folder_path,
                 t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                 t->tm_hour, t->tm_min, t->tm_sec);

        snprintf(thumb_path, sizeof(thumb_path),
                 "%s/THUMB_%04d%02d%02d_%02d%02d%02d.jpeg",
                 folder_path,
                 t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                 t->tm_hour, t->tm_min, t->tm_sec);

        // Check LDR and update mode before capture
        check_ldr_and_set_mode();

        // Update timestamp before capture (uses stored GPS coordinates)
        update_rgn_timestamp();
        usleep(80000);

        // Get main image from VENC channel 0
        s32Ret = RK_MPI_VENC_GetStream(0, &stFrameMain, 1000);
        if (s32Ret == RK_SUCCESS) {
            file = fopen(jpeg_path, "w");
            if (file) {
                pData = RK_MPI_MB_Handle2VirAddr(stFrameMain.pstPack->pMbBlk);
                if (pData) {
                    size_t written = fwrite(pData, 1, stFrameMain.pstPack->u32Len,
file);
                    fflush(file);
                    printf("☑ Main Image %d stored: %s (%zu bytes)\n", jpeg_id,
jpeg_path, written);
                    printf("⚲ GPS Location (from initialization): Lat=%s,
Lon=%s\n",
                           g_gps_latitude, g_gps_longitude);
                } else {
                    RK_LOGE("Failed to get virtual address for MB block");
                }
                fclose(file);
            }

            s32Ret = RK_MPI_VENC_ReleaseStream(0, &stFrameMain);
            if (s32Ret != RK_SUCCESS) {
                RK_LOGE("RK_MPI_VENC_ReleaseStream fail %x", s32Ret);
            }
        } else {
            RK_LOGE("RK_MPI_VENC_GetStream for main failed: 0x%x", s32Ret);
        }
```

```
        // Get thumbnail image from VENC channel 1
        s32Ret = RK_MPI_VENC_GetStream(1, &stFrameThumb, 1000);
        if (s32Ret == RK_SUCCESS) {
            thumb_file = fopen(thumb_path, "w");
            if (thumb_file) {
                pData = RK_MPI_MB_Handle2VirAddr(stFrameThumb.pstPack->pMbBlk);
                if (pData) {
                    size_t written = fwrite(pData, 1, stFrameThumb.pstPack->u32Len,
thumb_file);
                    fflush(thumb_file);
                    printf("☑ Thumbnail %d stored: %s (%zu bytes)\n", jpeg_id,
thumb_path, written);
                } else {
                    RK_LOGE("Failed to get virtual address for MB block (thumb)");
                }
                fclose(thumb_file);
            }

            s32Ret = RK_MPI_VENC_ReleaseStream(1, &stFrameThumb);
            if (s32Ret != RK_SUCCESS) {
                RK_LOGE("RK_MPI_VENC_ReleaseStream for thumb fail %x", s32Ret);
            }
        } else {
            RK_LOGE("RK_MPI_VENC_GetStream for thumb failed: 0x%x", s32Ret);
        }

        jpeg_id++;
        usleep(100000);
    }

    free(stFrameMain.pstPack);
    free(stFrameThumb.pstPack);
    return NULL;
}

// Helper to make directory safely (ignore if already exists)
static void make_dir_if_not_exists(const char *path)
{
    struct stat st = {0};
    if (stat(path, &st) == -1) {
        if (mkdir(path, 0777) == 0) {
            printf("[Folder Created] %s\n", path);
        } else {
            printf("[Warning] Failed to create folder %s: %s\n", path,
strerror(errno));
        }
    } else {
        // Folder already exists — optional to print this line
         printf("[Exists] %s\n", path);
    }
```

```
}

// VPSS initialization for both main and thumbnail channels
static RK_S32 test_vpss_init(int vpssGrp, int width, int height) {
    VPSS_GRP_ATTR_S stGrpAttr;
    VPSS_CHN_ATTR_S stChnAttr;
    RK_S32 ret;

    printf("========%s: Initializing VPSS group %d========\n", __func__, vpssGrp);

    memset(&stGrpAttr, 0, sizeof(stGrpAttr));
    stGrpAttr.u32MaxW = width;
    stGrpAttr.u32MaxH = height;
    stGrpAttr.enPixelFormat = RK_FMT_YUV420SP_VU;
    stGrpAttr.stFrameRate.s32SrcFrameRate = -1;
    stGrpAttr.stFrameRate.s32DstFrameRate = -1;

    ret = RK_MPI_VPSS_CreateGrp(vpssGrp, &stGrpAttr);
    if (ret != RK_SUCCESS) {
        RK_LOGE("RK_MPI_VPSS_CreateGrp failed: 0x%x", ret);
        return ret;
    }

    // Channel 0: Full size (2592x1944) - for main image
    memset(&stChnAttr, 0, sizeof(stChnAttr));
    stChnAttr.enChnMode = VPSS_CHN_MODE_USER;
    stChnAttr.enCompressMode = COMPRESS_MODE_NONE;
    stChnAttr.u32Width = width;
    stChnAttr.u32Height = height;
    stChnAttr.enPixelFormat = RK_FMT_YUV420SP_VU;
    stChnAttr.stFrameRate.s32SrcFrameRate = -1;
    stChnAttr.stFrameRate.s32DstFrameRate = -1;
    stChnAttr.u32Depth = 2;

    ret = RK_MPI_VPSS_SetChnAttr(vpssGrp, 0, &stChnAttr);
    if (ret != RK_SUCCESS) {
        RK_LOGE("RK_MPI_VPSS_SetChnAttr for chn0 failed: 0x%x", ret);
        RK_MPI_VPSS_DestroyGrp(vpssGrp);
        return ret;
    }

    ret = RK_MPI_VPSS_EnableChn(vpssGrp, 0);
    if (ret != RK_SUCCESS) {
        RK_LOGE("RK_MPI_VPSS_EnableChn for chn0 failed: 0x%x", ret);
        RK_MPI_VPSS_DestroyGrp(vpssGrp);
        return ret;
    }

    // Channel 1: Thumbnail size (320x240)
    memset(&stChnAttr, 0, sizeof(stChnAttr));
```

```
    stChnAttr.enChnMode = VPSS_CHN_MODE_USER;
    stChnAttr.enCompressMode = COMPRESS_MODE_NONE;
    stChnAttr.u32Width = 320;   // Thumbnail width
    stChnAttr.u32Height = 240;  // Thumbnail height
    stChnAttr.enPixelFormat = RK_FMT_YUV420SP_VU;
    stChnAttr.stFrameRate.s32SrcFrameRate = -1;
    stChnAttr.stFrameRate.s32DstFrameRate = -1;
    stChnAttr.u32Depth = 4;

    ret = RK_MPI_VPSS_SetChnAttr(vpssGrp, 1, &stChnAttr);
    if (ret != RK_SUCCESS) {
        RK_LOGE("RK_MPI_VPSS_SetChnAttr for chn1 failed: 0x%x", ret);
        RK_MPI_VPSS_DisableChn(vpssGrp, 0);
        RK_MPI_VPSS_DestroyGrp(vpssGrp);
        return ret;
    }

    ret = RK_MPI_VPSS_EnableChn(vpssGrp, 1);
    if (ret != RK_SUCCESS) {
        RK_LOGE("RK_MPI_VPSS_EnableChn for chn1 failed: 0x%x", ret);
        RK_MPI_VPSS_DisableChn(vpssGrp, 0);
        RK_MPI_VPSS_DestroyGrp(vpssGrp);
        return ret;
    }

    // Start VPSS group
    ret = RK_MPI_VPSS_StartGrp(vpssGrp);
    if (ret != RK_SUCCESS) {
        RK_LOGE("RK_MPI_VPSS_StartGrp failed: 0x%x", ret);
        RK_MPI_VPSS_DisableChn(vpssGrp, 1);
        RK_MPI_VPSS_DisableChn(vpssGrp, 0);
        RK_MPI_VPSS_DestroyGrp(vpssGrp);
        return ret;
    }

    printf("VPSS group %d initialized successfully\n", vpssGrp);
    printf("  - Channel 0: %dx%d (main image)\n", width, height);
    printf("  - Channel 1: 320x240 (thumbnail)\n");

    return RK_SUCCESS;
}

// Create /g_pOutPath/YYYY/MM/DD and return full path in folder_path
void create_full_date_folder(char *folder_path)
{
    time_t now = time(NULL);
    struct tm *t = localtime(&now);


    char year_path[MAX_PATH_LEN];
```

```
    char month_path[MAX_PATH_LEN];
    char base_path[MAX_PATH_LEN];

    memset(year_path, 0, sizeof(year_path));
    memset(month_path, 0, sizeof(month_path));
    memset(base_path, 0, sizeof(base_path));
    memset(folder_path, 0, sizeof(folder_path));

    // Ensure g_pOutPath ends with "/"
    if (g_pOutPath[strlen(g_pOutPath) - 1] == '/')
        snprintf(base_path, sizeof(base_path), "%s", g_pOutPath);
    else
        snprintf(base_path, sizeof(base_path), "%s/", g_pOutPath);

    // Year folder
    snprintf(year_path, sizeof(year_path), "%s%04d", base_path, t->tm_year + 1900);
    make_dir_if_not_exists(year_path);

    // Month folder
    snprintf(month_path, sizeof(month_path), "%s/%02d", year_path, t->tm_mon + 1);
    make_dir_if_not_exists(month_path);

    // Day folder (final)
    snprintf(folder_path, 256, "%s/%02d", month_path, t->tm_mday);
    make_dir_if_not_exists(folder_path);

    printf("[Active Save Path] %s\n", folder_path);
}


static RK_S32 test_venc_init(int chnId, int width, int height, RK_CODEC_ID_E enType)
{
    printf("========%s========\n", __func__);
    VENC_RECV_PIC_PARAM_S stRecvParam;
    VENC_CHN_ATTR_S stAttr;
    VENC_CHN_PARAM_S stParam;
    memset(&stAttr, 0, sizeof(VENC_CHN_ATTR_S));
    memset(&stParam, 0, sizeof(VENC_CHN_PARAM_S));

    stAttr.stVencAttr.enType = enType;
    stAttr.stVencAttr.enPixelFormat = RK_FMT_YUV420SP_VU;
    stAttr.stVencAttr.u32PicWidth = width;
    stAttr.stVencAttr.u32PicHeight = height;
    stAttr.stVencAttr.u32VirWidth = width;
    stAttr.stVencAttr.u32VirHeight = height;
    stAttr.stVencAttr.u32StreamBufCnt = 2;
    stAttr.stVencAttr.u32BufSize = width * height * 3 / 2;
    stAttr.stVencAttr.enMirror = MIRROR_NONE;

    stAttr.stVencAttr.stAttrJpege.bSupportDCF = RK_FALSE;
```

```
    stAttr.stVencAttr.stAttrJpege.stMPFCfg.u8LargeThumbNailNum = 0;
    stAttr.stVencAttr.stAttrJpege.enReceiveMode = VENC_PIC_RECEIVE_SINGLE;

    RK_MPI_VENC_CreateChn(chnId, &stAttr);

    stParam.stFrameRate.bEnable = RK_FALSE;
    stParam.stFrameRate.s32SrcFrmRateNum = 25;
    stParam.stFrameRate.s32SrcFrmRateDen = 1;
    stParam.stFrameRate.s32DstFrmRateNum = 10;
    stParam.stFrameRate.s32DstFrmRateDen = 1;
    RK_MPI_VENC_SetChnParam(chnId, &stParam);

    memset(&stRecvParam, 0, sizeof(VENC_RECV_PIC_PARAM_S));
    stRecvParam.s32RecvPicNum = 1;
    RK_MPI_VENC_StartRecvFrame(chnId, &stRecvParam);

    return 0;
}

// Add this function for thumbnail VENC
static RK_S32 test_venc_thumb_init(int chnId, int width, int height, RK_CODEC_ID_E
enType) {
    printf("========%s for thumbnail========\n", __func__);
    VENC_RECV_PIC_PARAM_S stRecvParam;
    VENC_CHN_ATTR_S stAttr;
    VENC_CHN_PARAM_S stParam;
    memset(&stAttr, 0, sizeof(VENC_CHN_ATTR_S));
    memset(&stParam, 0, sizeof(VENC_CHN_PARAM_S));

    stAttr.stVencAttr.enType = enType;
    stAttr.stVencAttr.enPixelFormat = RK_FMT_YUV420SP_VU;
    stAttr.stVencAttr.u32PicWidth = width;
    stAttr.stVencAttr.u32PicHeight = height;
    stAttr.stVencAttr.u32VirWidth = width;
    stAttr.stVencAttr.u32VirHeight = height;
    stAttr.stVencAttr.u32StreamBufCnt = 2;
    stAttr.stVencAttr.u32BufSize = width * height * 3 / 2;
    stAttr.stVencAttr.enMirror = MIRROR_NONE;

    // No thumbnail for thumbnail channel
    stAttr.stVencAttr.stAttrJpege.bSupportDCF = RK_FALSE;
    stAttr.stVencAttr.stAttrJpege.enReceiveMode = VENC_PIC_RECEIVE_SINGLE;

    RK_MPI_VENC_CreateChn(chnId, &stAttr);

    stParam.stFrameRate.bEnable = RK_FALSE;
    stParam.stFrameRate.s32SrcFrmRateNum = 25;
    stParam.stFrameRate.s32SrcFrmRateDen = 1;
    stParam.stFrameRate.s32DstFrmRateNum = 10;
    stParam.stFrameRate.s32DstFrmRateDen = 1;
```

```
    RK_MPI_VENC_SetChnParam(chnId, &stParam);

    memset(&stRecvParam, 0, sizeof(VENC_RECV_PIC_PARAM_S));
    stRecvParam.s32RecvPicNum = 1;
    RK_MPI_VENC_StartRecvFrame(chnId, &stRecvParam);

    return 0;
}

int vi_dev_init(void) {
    printf("%s\n", __func__);
    int ret = 0;
    int devId = 0;
    int pipeId = devId;

    VI_DEV_ATTR_S stDevAttr;
    VI_DEV_BIND_PIPE_S stBindPipe;
    memset(&stDevAttr, 0, sizeof(stDevAttr));
    memset(&stBindPipe, 0, sizeof(stBindPipe));

    ret = RK_MPI_VI_GetDevAttr(devId, &stDevAttr);
    if (ret == RK_ERR_VI_NOT_CONFIG) {
        ret = RK_MPI_VI_SetDevAttr(devId, &stDevAttr);
        if (ret != RK_SUCCESS) {
            printf("RK_MPI_VI_SetDevAttr %x\n", ret);
            return -1;
        }
    } else {
        printf("RK_MPI_VI_SetDevAttr already\n");
    }

    ret = RK_MPI_VI_GetDevIsEnable(devId);
    if (ret != RK_SUCCESS) {
        ret = RK_MPI_VI_EnableDev(devId);
        if (ret != RK_SUCCESS) {
            printf("RK_MPI_VI_EnableDev %x\n", ret);
            return -1;
        }
        stBindPipe.u32Num = 1;
        stBindPipe.PipeId[0] = pipeId;
        ret = RK_MPI_VI_SetDevBindPipe(devId, &stBindPipe);
        if (ret != RK_SUCCESS) {
            printf("RK_MPI_VI_SetDevBindPipe %x\n", ret);
            return -1;
        }
    } else {
        printf("RK_MPI_VI_EnableDev already\n");
    }

    return 0;
```

```
}

int vi_chn_init(int channelId, int width, int height) {
    int ret;
    int buf_cnt = 2;

    VI_CHN_ATTR_S vi_chn_attr;
    memset(&vi_chn_attr, 0, sizeof(vi_chn_attr));
    vi_chn_attr.stIspOpt.u32BufCount = buf_cnt;
    vi_chn_attr.stIspOpt.enMemoryType = VI_V4L2_MEMORY_TYPE_DMABUF;
    vi_chn_attr.stSize.u32Width = width;
    vi_chn_attr.stSize.u32Height = height;
    vi_chn_attr.enPixelFormat = RK_FMT_YUV420SP_VU;
    vi_chn_attr.enCompressMode = COMPRESS_MODE_NONE;
    vi_chn_attr.u32Depth = 0;

    ret = RK_MPI_VI_SetChnAttr(0, channelId, &vi_chn_attr);
    ret |= RK_MPI_VI_EnableChn(0, channelId);
    if (ret) {
        printf("ERROR: create VI error! ret=%d\n", ret);
        return ret;
    }

    return ret;
}

void gpiointi(void) {
    // LDR INIT
    system("echo 72 > /sys/class/gpio/export");
    system("echo in > /sys/class/gpio/gpio72/direction");
    system("echo 53 > /sys/class/gpio/export");
    system("echo out > /sys/class/gpio/gpio53/direction");
    system("echo 0 > /sys/class/gpio/gpio53/value");
    system("echo 52 > /sys/class/gpio/export");
    system("echo out > /sys/class/gpio/gpio52/direction");
    system("echo 0 > /sys/class/gpio/gpio52/value");
}

void getimage(void) {
    VENC_RECV_PIC_PARAM_S stRecvParam;
    usleep(400000);
    if (x==1)
    {
    getTimeRTC();            // Your existing function
    setRTCAlarm(1,0);
    x=2;
    }
    // Check LDR and update mode before capture
    check_ldr_and_set_mode();
```

```
    int ldr_fd = open("/sys/class/gpio/gpio72/value", O_RDONLY);
    if (ldr_fd >= 0) {
        char val;
        if (read(ldr_fd, &val, 1) > 0) {
            if (val == '0') {
                printf("LDR says: DAY - Using BLACK text\n");
                close(ldr_fd);
            } else {
                printf("LDR says: NIGHT - Using WHITE text\n");
                system("echo 1 > /sys/class/gpio/gpio53/value");
            }
        }
    } else {
        perror("Failed to read LDR GPIO");
    }
    // Capture
    memset(&stRecvParam, 0, sizeof(stRecvParam));
    stRecvParam.s32RecvPicNum = 1;
    RK_MPI_VENC_StartRecvFrame(0, &stRecvParam);
    usleep(30000);
    RK_MPI_VENC_StartRecvFrame(1, &stRecvParam);
    usleep(30000);

    system("echo 0 > /sys/class/gpio/gpio53/value");
    system("echo mem > /sys/power/state");
}

void getimagebustmode(void) {
    VENC_RECV_PIC_PARAM_S stRecvParam;
    usleep(400000);

    // Check LDR and update mode before capture
    check_ldr_and_set_mode();

    int ldr_fd = open("/sys/class/gpio/gpio72/value", O_RDONLY);
    if (ldr_fd >= 0) {
        char val;
        if (read(ldr_fd, &val, 1) > 0) {
            if (val == '0') {
                printf("LDR says: DAY \n");
                close(ldr_fd);
            } else {
                printf("LDR says: NIGHT\n");
                system("echo 1 > /sys/class/gpio/gpio53/value");
            }
        }
    } else {
        perror("Failed to read LDR GPIO");
    }
```

```
    // Capture
    memset(&stRecvParam, 0, sizeof(stRecvParam));
    stRecvParam.s32RecvPicNum = 3;
    RK_MPI_VENC_StartRecvFrame(0, &stRecvParam);
    usleep(30000);
    RK_MPI_VENC_StartRecvFrame(1, &stRecvParam);
    usleep(30000);

    system("echo 0 > /sys/class/gpio/gpio53/value");
    system("echo mem > /sys/power/state");
}

void handle_volumeup_event(struct input_event ev) {
    if (ev.value != 1) return;

    static struct timeval last_press = {0,0};
    static int press_count = 0;
    static bool double_press_handled = false;

    struct timeval now = ev.time;
    long diff_ms = (now.tv_sec - last_press.tv_sec) * 1000 +
                   (now.tv_usec - last_press.tv_usec) / 1000;

    if (diff_ms > 500) {
        press_count = 0;
        double_press_handled = false;
    }

    press_count++;
    last_press = now;

    if (press_count == 1) {
        printf("Single press registered\n");
        B=1;
    }
    else if (press_count == 2 && !double_press_handled) {
        printf("Double press: capture 3 images\n");
        B = 3;
        system("echo 1 > /sys/class/gpio/gpio52/value");
        usleep(10000);
        system("echo 0 > /sys/class/gpio/gpio52/value");
        double_press_handled = true;
        press_count = 0;
    }
}

int getTimeRTC(void)
{
    int file;
    const char *i2c_device = "/dev/i2c-3";  // I2C bus 3
```

```
    int addr = 0x32;                        // External RTC I2C address

    // Registers verified from your test
    unsigned char reg_sec   = 0x11;
    unsigned char reg_min   = 0x12;
    unsigned char reg_hour  = 0x13;
    unsigned char reg_day   = 0x15;
    unsigned char reg_month = 0x16;
    unsigned char reg_year  = 0x17;

    unsigned char sec, min, hour, day, month, year;

    // Open I2C bus
    if ((file = open(i2c_device, O_RDWR)) < 0) {
        perror("Failed to open I2C device");
        return 1;
    }

    // Set I2C slave address
    if (ioctl(file, I2C_SLAVE, addr) < 0) {
        perror("Failed to set I2C address");
        close(file);
        return 1;
    }

#define READ_REG(reg, var) \
    do { \
        if (write(file, &reg, 1) != 1 || read(file, &var, 1) != 1) { \
            perror("Failed to read register"); \
            close(file); \
            return 1; \
        } \
    } while (0)

    // Read time registers
    READ_REG(reg_sec, sec);
    READ_REG(reg_min, min);
    READ_REG(reg_hour, hour);
    READ_REG(reg_day, day);
    READ_REG(reg_month, month);
    READ_REG(reg_year, year);

    close(file);

    // Convert BCD → decimal
    int dec_sec   = bcd_to_dec(sec & 0x7F);
    int dec_min   = bcd_to_dec(min & 0x7F);
    int dec_hour  = bcd_to_dec(hour & 0x3F);
    int dec_day   = bcd_to_dec(day & 0x3F);
    int dec_month = bcd_to_dec(month & 0x1F);
```

```
    int dec_year  = bcd_to_dec(year);

    printf("Current RTC Date: 20%02d-%02d-%02d\n", dec_year, dec_month, dec_day);
    printf("Current RTC Time: %02d:%02d:%02d\n", dec_hour, dec_min, dec_sec);

    int fd = open("/dev/rtc0", O_RDWR);
    if (fd >= 0) {
     struct rtc_time rtc_tm;
     rtc_tm.tm_sec = dec_sec;
     rtc_tm.tm_min = dec_min;
     rtc_tm.tm_hour = dec_hour;
     rtc_tm.tm_mday = dec_day;
     rtc_tm.tm_mon = dec_month - 1;
     rtc_tm.tm_year = (dec_year + 2000) - 1900;
    if (ioctl(fd, RTC_SET_TIME, &rtc_tm) < 0) {
        perror("Failed to set /dev/rtc0");
    } else {
        printf("☑ /dev/rtc0 updated from external RTC\n");
    }
    close(fd);
    } else {
         perror("Failed to open /dev/rtc0");
 }


    return 0;
}

// Sets an RTC alarm X hours and Y minutes ahead of current time
int setRTCAlarm(int alarm_hour, int add_minutes)
{
    const char *i2c_device = "/dev/i2c-3";
    int addr = 0x32;  // External RTC I2C address
    int file;

    unsigned char reg_min   = 0x12;
    unsigned char reg_hour  = 0x13;
    unsigned char reg_alm_min = 0x18;
    unsigned char reg_alm_hour = 0x19;
    unsigned char reg_status = 0x1E;
    unsigned char reg_control = 0x1F;

    unsigned char min_bcd, hour_bcd;
    int dec_min, dec_hour;

    // Open I2C device
    if ((file = open(i2c_device, O_RDWR)) < 0) {
        perror("Failed to open I2C bus");
        return 1;
    }
```

```
if (ioctl(file, I2C_SLAVE, addr) < 0) {
    perror("Failed to set I2C address");
    close(file);
    return 1;
}

// ---- Read current hour and minute ----
if (write(file, &reg_min, 1) != 1 || read(file, &min_bcd, 1) != 1) {
    perror("Failed to read minute register");
    close(file);
    return 1;
}
if (write(file, &reg_hour, 1) != 1 || read(file, &hour_bcd, 1) != 1) {
    perror("Failed to read hour register");
    close(file);
    return 1;
}

dec_min = bcd_to_dec(min_bcd & 0x7F);
dec_hour = bcd_to_dec(hour_bcd & 0x3F);

printf("Current RTC time: %02d:%02d\n", dec_hour, dec_min);

// ---- Add hours if needed ----
if (alarm_hour > 0) {
    dec_hour += alarm_hour;
    if (dec_hour >= 24)
        dec_hour -= 24;
}

// ---- Add minutes ----
dec_min += add_minutes;
if (dec_min >= 60) {
    dec_min -= 60;
    dec_hour++;
    if (dec_hour >= 24)
        dec_hour = 0;
}

// ---- Convert to BCD ----
unsigned char new_min_bcd = dec_to_bcd(dec_min);
unsigned char new_hour_bcd = dec_to_bcd(dec_hour);

// ---- Program alarm ----
unsigned char buffer[2];

// Write Alarm Minute
buffer[0] = reg_alm_min;
buffer[1] = new_min_bcd;
```

```
    if (write(file, buffer, 2) != 2) {
        perror("Failed to write alarm minute");
        close(file);
        return 1;
    }

    // Write Alarm Hour
    buffer[0] = reg_alm_hour;
    buffer[1] = new_hour_bcd;
    if (write(file, buffer, 2) != 2) {
        perror("Failed to write alarm hour");
        close(file);
        return 1;
    }

    // Clear previous alarm flag
    buffer[0] = reg_status;
    buffer[1] = 0x00;
    if (write(file, buffer, 2) != 2) {
        perror("Failed to clear alarm flag");
        close(file);
        return 1;
    }

    // Enable alarm interrupt
    buffer[0] = reg_control;
    buffer[1] = 0x08;  // Enable alarm interrupt only
    if (write(file, buffer, 2) != 2) {
        perror("Failed to enable alarm interrupt");
        close(file);
        return 1;
    }

    printf("Alarm set for %02d:%02d (in %d hours and %d minutes)\n", dec_hour,
dec_min, alarm_hour, add_minutes);

    close(file);
    return 0;
}



// Convert BCD to decimal
int bcd_to_dec(unsigned char val) {
    return ((val >> 4) * 10) + (val & 0x0F);
}

// Convert decimal to BCD
unsigned char dec_to_bcd(int val) {
    return ((val / 10) << 4) | (val % 10);
```

```
}


int main(int argc, char *argv[]) {
    freopen("/mnt/sdcard/capture.log", "a+", stdout);
    freopen("/mnt/sdcard/capture.log", "a+", stderr);
    setbuf(stdout, NULL);

    setenv("RKAIQ_XML_DIR", "/oem/etc/iqfiles", 1);
    system("rkaiq_3A_server &");
    usleep(200000);

    RK_S32 s32Ret = RK_FAILURE;
    RK_U32 u32Width = 2592;
    RK_U32 u32Height = 1944;
    RK_CODEC_ID_E enCodecType = RK_VIDEO_ID_JPEG;
    RK_CHAR *pCodecName = "JPEG";
    RK_S32 s32chnlId = 0;
    int c;
    int ret = -1;

    while ((c = getopt(argc, argv, optstr)) != -1) {
        switch (c) {
        case 'w':
            u32Width = atoi(optarg);
            break;
        case 'h':
            u32Height = atoi(optarg);
            break;
        case 'I':
            s32chnlId = atoi(optarg);
            break;
        case '?':
        default:
            print_usage(argv[0]);
            return -1;
        }
    }

    printf("#CodecName:%s\n", pCodecName);
    printf("#Resolution: %dx%d\n", u32Width, u32Height);
    printf("#CameraIdx: %d\n\n", s32chnlId);

    signal(SIGINT, sigterm_handler);

    if (RK_MPI_SYS_Init() != RK_SUCCESS) {
        RK_LOGE("rk mpi sys init fail!");
        goto __FAILED;
    }
    gpiointi();
```

```
    // Initialize GPS module ONLY ONCE during startup with 9600 baud
    printf("=== INITIALIZING GPS MODULE (ONE-TIME) at 9600 baud ===\n");
    if (!initialize_gps_module()) {
        printf("WARNING: GPS initialization failed, continuing without GPS\n");
        printf("GPS coordinates will show as: N/A\n");
    } else {
        printf("=== GPS INITIALIZATION COMPLETE ===\n");
        printf("Coordinates obtained: Lat=%s, Lon=%s\n", g_gps_latitude,
g_gps_longitude);
        printf("These coordinates will be used for ALL captured images\n");
    }

    // Initialize components in correct order
    vi_dev_init();
    vi_chn_init(s32chnlId, u32Width, u32Height);
    test_vpss_init(0, u32Width, u32Height);

    // Add delay between VENC creations
    test_venc_init(0, u32Width, u32Height, enCodecType);
    usleep(100000); // 100ms delay
    test_venc_thumb_init(1, 320, 240, enCodecType);

    // Initialize adaptive timestamp region
    init_timestamp_rgn(u32Width, u32Height);

    // Bind VI to VPSS
    MPP_CHN_S stSrcChn, stDestChn;
    stSrcChn.enModId = RK_ID_VI;
    stSrcChn.s32DevId = 0;
    stSrcChn.s32ChnId = s32chnlId;

    stDestChn.enModId = RK_ID_VPSS;
    stDestChn.s32DevId = 0;
    stDestChn.s32ChnId = 0;

    printf("====RK_MPI_SYS_Bind vi0 to vpss0====\n");
    s32Ret = RK_MPI_SYS_Bind(&stSrcChn, &stDestChn);
    if (s32Ret != RK_SUCCESS) {
        RK_LOGE("bind vi0 to vpss0 failed");
        goto __FAILED;
    }

    // Bind VPSS channel 0 to VENC channel 0 (main image)
    stSrcChn.enModId = RK_ID_VPSS;
    stSrcChn.s32DevId = 0;
    stSrcChn.s32ChnId = 0;

    stDestChn.enModId = RK_ID_VENC;
    stDestChn.s32DevId = 0;
    stDestChn.s32ChnId = 0;
```

```
printf("====RK_MPI_SYS_Bind vpss0 to venc0====\n");
s32Ret = RK_MPI_SYS_Bind(&stSrcChn, &stDestChn);
if (s32Ret != RK_SUCCESS) {
    RK_LOGE("bind vpss0 to venc0 failed");
    goto __FAILED;
}

// Bind VPSS channel 1 to VENC channel 1 (thumbnail)
stSrcChn.enModId = RK_ID_VPSS;
stSrcChn.s32DevId = 0;
stSrcChn.s32ChnId = 1;

stDestChn.enModId = RK_ID_VENC;
stDestChn.s32DevId = 0;
stDestChn.s32ChnId = 1;

printf("====RK_MPI_SYS_Bind vpss1 to venc1====\n");
s32Ret = RK_MPI_SYS_Bind(&stSrcChn, &stDestChn);
if (s32Ret != RK_SUCCESS) {
    RK_LOGE("bind vpss1 to venc1 failed");
    goto __FAILED;
}

int gpio_fd = open("/dev/input/event1", O_RDONLY);
if (gpio_fd < 0) {
    perror("Failed to open input event");
    return -1;
}


getTimeRTC();


pthread_t main_thread;
pthread_create(&main_thread, NULL, GetMediaBuffer0, NULL);

VENC_JPEG_PARAM_S stJpegParam;
memset(&stJpegParam, 0, sizeof(stJpegParam));
stJpegParam.u32Qfactor = 99;
RK_MPI_VENC_SetJpegParam(0, &stJpegParam);

struct input_event ev;

while (1) {
    if (read(gpio_fd, &ev, sizeof(ev)) < (int)sizeof(ev)) {
        perror("read");
        break;
    }
```

```
    if (ev.type == EV_KEY && ev.value == 1) {
        switch (ev.code) {
            case KEY_WAKEUP:
                printf("WAKEUP interrupt (PIR)\n");
                if(B == 3) {
                    printf("burst mode is ON\n");
                    getimagebustmode();
                } else {
                    getTimeRTC();
                    getimage();
                }
                break;

            case KEY_POWER:
                printf("BUTTON pressed\n");
                system("echo 1 > /sys/class/gpio/gpio53/value");
                usleep(200000);
                system("echo 0 > /sys/class/gpio/gpio53/value");
                system("echo mem > /sys/power/state");
                break;

            case KEY_VOLUMEUP:
                printf("115:KEY_VOLUMEUP(pin number 0\n");
                getTimeRTC();            // Your existing function
                setRTCAlarm(1,0);
                if(B == 3) {
                    printf("burst mode is ON\n");
                    getimagebustmode();
                } else {
                    getimage();
                }

                //handle_volumeup_event(ev);
                //system("echo mem > /sys/power/state");
                break;

            case KEY_VOLUMEDOWN:
                printf("114:KEY_VOLUMEDOWN\n");
                system("echo mem > /sys/power/state");
                break;

            default:
                printf("Unknown key code %d\n", ev.code);
                break;
        }
    }
}

sleep(1);
pthread_join(main_thread, NULL);
```

```
    // Cleanup - unbind in reverse order
    // Unbind VPSS1 from VENC1
    stSrcChn.enModId = RK_ID_VPSS;
    stSrcChn.s32DevId = 0;
    stSrcChn.s32ChnId = 1;
    stDestChn.enModId = RK_ID_VENC;
    stDestChn.s32DevId = 0;
    stDestChn.s32ChnId = 1;
    RK_MPI_SYS_UnBind(&stSrcChn, &stDestChn);

    // Unbind VPSS0 from VENC0
    stSrcChn.enModId = RK_ID_VPSS;
    stSrcChn.s32DevId = 0;
    stSrcChn.s32ChnId = 0;
    stDestChn.enModId = RK_ID_VENC;
    stDestChn.s32DevId = 0;
    stDestChn.s32ChnId = 0;
    RK_MPI_SYS_UnBind(&stSrcChn, &stDestChn);

    // Unbind VI from VPSS
    stSrcChn.enModId = RK_ID_VI;
    stSrcChn.s32DevId = 0;
    stSrcChn.s32ChnId = s32chnlId;
    stDestChn.enModId = RK_ID_VPSS;
    stDestChn.s32DevId = 0;
    stDestChn.s32ChnId = 0;
    RK_MPI_SYS_UnBind(&stSrcChn, &stDestChn);

    RK_MPI_RGN_Destroy(g_rgn_handle);
    RK_MPI_VENC_StopRecvFrame(0);
    RK_MPI_VENC_StopRecvFrame(1);
    RK_MPI_VENC_DestroyChn(0);
    RK_MPI_VENC_DestroyChn(1);
    RK_MPI_VPSS_StopGrp(0);
    RK_MPI_VPSS_DisableChn(0, 1);
    RK_MPI_VPSS_DisableChn(0, 0);
    RK_MPI_VPSS_DestroyGrp(0);
    RK_MPI_VI_DisableChn(0, s32chnlId);
    RK_MPI_VI_DisableDev(0);

    pthread_mutex_destroy(&g_rgn_mutex);
    pthread_mutex_destroy(&g_gps_mutex);
    system("killall rkaiq_3A_server");

__FAILED:
    RK_MPI_SYS_Exit();
    return ret;
}
```