

自然写智能点阵笔嵌入式固件软件 V1.0

软件著作权鉴别材料 — 源程序

权利人：深圳自然写科技有限公司

版本号：V1.0

源程序目录结构

```
12-writech-pen-firmware/  
├── main.c  
├── cache/  
│   └── offline_storage.c  
├── codec/  
│   └── dot_decoder.c  
├── driver/  
│   ├── camera_driver.c  
│   └── pressure_sensor.c  
├── power/  
│   └── power_manager.c  
└── task/  
    ├── ble_send_task.c  
    ├── coordinate_task.c  
    ├── image_capture_task.c  
    └── power_monitor_task.c
```

源程序文件清单

(根目录)

main.c

```
/*  
 * 自然写智能点阵笔嵌入式固件软件 V1.0  
 * main.c - 主函数入口与RTOS任务创建  
 *  
 * 功能说明:
```

```

* 1. 系统硬件初始化 (GPIO/SPI/I2C/ADC/DMA)
* 2. FreeRTOS内核启动与任务创建
* 3. 各功能模块初始化协调
* 4. 看门狗定时器配置
* 5. 系统错误处理与故障恢复
*
* 硬件平台: ARM Cortex-M4F MCU
* RTOS: FreeRTOS 10.x
*/

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <stdio.h>

/* FreeRTOS头文件 */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "timers.h"
#include "event_groups.h"

/* 硬件抽象层头文件 */
#include "hal_gpio.h"
#include "hal_spi.h"
#include "hal_i2c.h"
#include "hal_adc.h"
#include "hal_dma.h"
#include "hal_wdt.h"
#include "hal_flash.h"
#include "hal_rtc.h"

/* 功能模块头文件 */
#include "camera_driver.h"
#include "pressure_sensor.h"
#include "led_driver.h"
#include "ble_gatt_server.h"
#include "dot_decoder.h"
#include "power_manager.h"
#include "offline_storage.h"

/* ===== 任务优先级定义 ===== */

/* 图像采集任务（最高优先级，需要精确的100Hz定时） */
#define TASK_IMAGE_CAPTURE_PRIORITY    (configMAX_PRIORITIES - 1)

/* 坐标计算任务 */
#define TASK_COORDINATE_PRIORITY       (configMAX_PRIORITIES - 2)

/* BLE发送任务 */
#define TASK_BLE_SEND_PRIORITY         (configMAX_PRIORITIES - 3)

/* 电源监测任务（较低优先级） */
#define TASK_POWER_MONITOR_PRIORITY    (tskIDLE_PRIORITY + 2)

/* 看门狗喂狗任务 */

```

```

#define TASK_WATCHDOG_PRIORITY          (tskIDLE_PRIORITY + 1)

/* ===== 任务栈大小定义 (单位: 字) ===== */

#define TASK_IMAGE_CAPTURE_STACK_SIZE   512
#define TASK_COORDINATE_STACK_SIZE      1024
#define TASK_BLE_SEND_STACK_SIZE        512
#define TASK_POWER_MONITOR_STACK_SIZE   256
#define TASK_WATCHDOG_STACK_SIZE        128

/* ===== 全局队列与信号量 ===== */

/* 图像数据队列 (采集任务 → 坐标计算任务) */
QueueHandle_t g_image_data_queue;

/* 坐标数据队列 (坐标计算 → BLE发送) */
QueueHandle_t g_coordinate_queue;

/* BLE连接状态事件组 */
EventGroupHandle_t g_ble_event_group;

/* 系统状态互斥锁 */
SemaphoreHandle_t g_system_mutex;

/* ===== 事件位定义 ===== */
#define EVT_BLE_CONNECTED                (1 << 0)
#define EVT_BLE_DISCONNECTED             (1 << 1)
#define EVT_PEN_DOWN                     (1 << 2)
#define EVT_PEN_UP                       (1 << 3)
#define EVT_LOW_BATTERY                   (1 << 4)
#define EVT_CHARGING                      (1 << 5)
#define EVT_OTA_START                     (1 << 6)

/* ===== 全局系统状态 ===== */

typedef struct {
    bool pen_is_down;                /* 笔尖是否接触纸面 */
    bool ble_connected;              /* BLE是否已连接 */
    bool is_charging;                /* 是否正在充电 */
    uint8_t battery_percent;         /* 电量百分比 */
    uint32_t total_strokes;          /* 累计笔画数 */
    uint32_t uptime_seconds;         /* 运行时长 */
    uint8_t error_flags;             /* 错误标志位 */
} SystemState;

static SystemState g_system_state;

/* ===== 任务句柄 ===== */

static TaskHandle_t g_task_image_capture;
static TaskHandle_t g_task_coordinate;
static TaskHandle_t g_task_ble_send;
static TaskHandle_t g_task_power_monitor;
static TaskHandle_t g_task_watchdog;

/* ===== 函数前向声明 ===== */

```

```

static void hardware_init(void);
static void create_rtos_objects(void);
static void create_tasks(void);
static void watchdog_task(void *pvParameters);

/* 外部任务函数（各功能模块中实现） */
extern void image_capture_task(void *pvParameters);
extern void coordinate_task(void *pvParameters);
extern void ble_send_task(void *pvParameters);
extern void power_monitor_task(void *pvParameters);

/* ===== 主函数 ===== */

/**
 * 系统入口点
 * 完成硬件初始化后启动FreeRTOS调度器
 */
int main(void) {
    /* 步骤1: 系统时钟配置 (PLL → 168MHz) */
    SystemClock_Config();

    /* 步骤2: 基础硬件初始化 */
    hardware_init();

    /* 步骤3: LED指示启动中 (蓝色闪烁) */
    led_set_mode(LED_MODE_BLINK_BLUE);

    /* 步骤4: 初始化全局状态 */
    memset(&g_system_state, 0, sizeof(g_system_state));

    /* 步骤5: 创建RTOS同步对象 */
    create_rtos_objects();

    /* 步骤6: 创建功能任务 */
    create_tasks();

    /* 步骤7: 启动看门狗定时器 (超时8秒) */
    hal_wdt_init(8000);
    hal_wdt_start();

    /* 步骤8: 启动FreeRTOS调度器 (不应返回) */
    vTaskStartScheduler();

    /* 如果到达这里说明调度器启动失败 */
    led_set_mode(LED_MODE_SOLID_RED);
    while (1) {
        /* 系统错误, 死循环 */
    }

    return 0;
}

/* ===== 硬件初始化 ===== */

/**
 * 初始化所有硬件外设
 */

```

```

static void hardware_init(void) {
    /* GPIO初始化（笔尖接触检测引脚、充电检测引脚） */
    hal_gpio_init();

    /* SPI初始化（连接CMOS图像传感器，主模式 8MHz） */
    hal_spi_init(SPI_PORT_1, SPI_MODE_MASTER, 8000000);

    /* I2C初始化（连接压力传感器和IMU） */
    hal_i2c_init(I2C_PORT_1, 400000); /* 400kHz快速模式 */

    /* ADC初始化（电池电压检测，12位分辨率） */
    hal_adc_init(ADC_CHANNEL_BATTERY, ADC_RESOLUTION_12BIT);

    /* DMA初始化（SPI图像数据DMA传输） */
    hal_dma_init(DMA_CHANNEL_SPI_RX);

    /* Flash初始化（离线缓存存储） */
    hal_flash_init();

    /* RTC初始化（时间戳生成） */
    hal_rtc_init();

    /* 摄像头传感器初始化 */
    camera_driver_init();

    /* 压力传感器校准 */
    pressure_sensor_init();
    pressure_sensor_calibrate();

    /* LED驱动初始化 */
    led_driver_init();

    /* BLE协议栈初始化 */
    ble_gatt_server_init();

    /* 点阵码解码器初始化 */
    dot_decoder_init();

    /* 电源管理初始化 */
    power_manager_init();

    /* 离线存储初始化 */
    offline_storage_init();
}

/* ===== RTOS对象创建 ===== */

/**
 * 创建队列、信号量、事件组等RTOS同步对象
 */
static void create_rtos_objects(void) {
    /*
     * 图像数据队列：采集任务以100Hz频率产生数据
     * 队列深度10帧，每帧包含图像元数据（不含原始像素）
     */
    g_image_data_queue = xQueueCreate(10, sizeof(ImageFrameMetadata));
    configASSERT(g_image_data_queue != NULL);
}

```

```

/*
 * 坐标数据队列：坐标计算结果 → BLE发送
 * 队列深度20，容纳突发计算结果
 */
g_coordinate_queue = xQueueCreate(20, sizeof(CoordinatePacket));
configASSERT(g_coordinate_queue != NULL);

/* BLE事件组 */
g_ble_event_group = xEventGroupCreate();
configASSERT(g_ble_event_group != NULL);

/* 系统状态互斥锁 */
g_system_mutex = xSemaphoreCreateMutex();
configASSERT(g_system_mutex != NULL);
}

/* ===== 任务创建 ===== */

/**
 * 创建所有FreeRTOS任务
 */
static void create_tasks(void) {
    BaseType_t ret;

    /* 图像采集任务（100Hz定时采集CMOS图像） */
    ret = xTaskCreate(image_capture_task, "ImgCap",
                     TASK_IMAGE_CAPTURE_STACK_SIZE, NULL,
                     TASK_IMAGE_CAPTURE_PRIORITY, &g_task_image_capture);
    configASSERT(ret == pdPASS);

    /* 坐标计算任务（点阵码解码+坐标计算） */
    ret = xTaskCreate(coordinate_task, "CoordCalc",
                     TASK_COORDINATE_STACK_SIZE, NULL,
                     TASK_COORDINATE_PRIORITY, &g_task_coordinate);
    configASSERT(ret == pdPASS);

    /* BLE发送任务（坐标数据打包+BLE通知发送） */
    ret = xTaskCreate(ble_send_task, "BLESend",
                     TASK_BLE_SEND_STACK_SIZE, NULL,
                     TASK_BLE_SEND_PRIORITY, &g_task_ble_send);
    configASSERT(ret == pdPASS);

    /* 电源监测任务（电池电压/充电状态/低功耗管理） */
    ret = xTaskCreate(power_monitor_task, "PwrMon",
                     TASK_POWER_MONITOR_STACK_SIZE, NULL,
                     TASK_POWER_MONITOR_PRIORITY, &g_task_power_monitor);
    configASSERT(ret == pdPASS);

    /* 看门狗喂狗任务 */
    ret = xTaskCreate(watchdog_task, "WDT",
                     TASK_WATCHDOG_STACK_SIZE, NULL,
                     TASK_WATCHDOG_PRIORITY, &g_task_watchdog);
    configASSERT(ret == pdPASS);
}

/* ===== 看门狗任务 ===== */

```

```

/**
 * 看门狗喂狗任务
 * 周期性喂狗，防止系统死锁导致的假死
 * 如果各功能任务异常停止，看门狗将触发系统复位
 */
static void watchdog_task(void *pvParameters) {
    (void)pvParameters;

    TickType_t last_wake_time = xTaskGetTickCount();

    while (1) {
        /* 每2秒喂一次狗（看门狗超时8秒，留足余量） */
        hal_wdt_feed();

        /* 更新运行时长 */
        if (xSemaphoreTake(g_system_mutex, pdMS_TO_TICKS(100)) == pdTRUE) {
            g_system_state.uptime_seconds += 2;
            xSemaphoreGive(g_system_mutex);
        }

        /* 检查各任务是否正常运行 */
        if (eTaskGetState(g_task_image_capture) == eSuspended &&
            g_system_state.pen_is_down) {
            /* 图像采集任务异常挂起但笔在书写，尝试恢复 */
            vTaskResume(g_task_image_capture);
        }

        vTaskDelayUntil(&last_wake_time, pdMS_TO_TICKS(2000));
    }
}

/* ===== FreeRTOS回调函数 ===== */

/**
 * 栈溢出钩子函数
 * 任何任务发生栈溢出时被调用
 */
void vApplicationStackOverflowHook(TaskHandle_t xTask, char *pcTaskName) {
    /* 记录错误信息到Flash */
    g_system_state.error_flags |= 0x01;

    /* LED红色快闪指示严重错误 */
    led_set_mode(LED_MODE_FAST_BLINK_RED);

    /* 触发系统复位 */
    hal_system_reset();
}

/**
 * Malloc失败钩子函数
 */
void vApplicationMallocFailedHook(void) {
    g_system_state.error_flags |= 0x02;
    led_set_mode(LED_MODE_FAST_BLINK_RED);
    hal_system_reset();
}

```

```

/**
 * 空闲任务钩子函数
 * 在CPU空闲时进入低功耗模式以节省电量
 */
void vApplicationIdleHook(void) {
    /* 如果笔没有在书写且BLE空闲，进入轻度睡眠 */
    if (!g_system_state.pen_is_down && !g_system_state.ble_connected) {
        power_enter_light_sleep();
    }
}

```

cache/

cache/offline_storage.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * offline_storage.c - 离线Flash缓存存储
 *
 * 功能说明：
 * 1. 在BLE断连时将笔迹数据缓存到外部SPI Flash
 * 2. BLE重新连接后自动回传缓存数据
 * 3. 环形缓冲区管理Flash存储空间
 * 4. 掉电安全的写入机制（写入前擦除校验）
 * 5. 存储使用统计与容量告警
 */

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

#include "hal_flash.h"

/* ===== Flash存储参数 ===== */

/* 外部SPI Flash总容量（4MB） */
#define FLASH_TOTAL_SIZE      (4 * 1024 * 1024)

/* Flash扇区大小（4KB，最小擦除单元） */
#define FLASH_SECTOR_SIZE     4096

/* Flash页大小（256字节，最小写入单元） */
#define FLASH_PAGE_SIZE       256

/* 离线存储起始地址（前64KB保留给系统配置） */
#define STORAGE_START_ADDR    (64 * 1024)

/* 离线存储可用大小 */
#define STORAGE_AVAILABLE      (FLASH_TOTAL_SIZE - STORAGE_START_ADDR)

/* 可用扇区数量 */
#define STORAGE_SECTOR_COUNT   (STORAGE_AVAILABLE / FLASH_SECTOR_SIZE)

```



```

/* 每条笔迹记录大小（固定长度，便于管理） */
#define RECORD_SIZE                16

/* 每个扇区能存储的记录数 */
#define RECORDS_PER_SECTOR        (FLASH_SECTOR_SIZE / RECORD_SIZE)

/* 记录头标识 */
#define RECORD_MAGIC                0xAB

/* ===== 数据结构 ===== */

/* 存储记录结构（16字节固定长度） */
typedef struct __attribute__((packed)) {
    uint8_t  magic;                /* 记录标识 0xAB */
    uint8_t  record_type;          /* 记录类型：0=坐标，1=笔落下，2=笔抬起 */
    uint32_t x;                    /* X坐标 */
    uint32_t y;                    /* Y坐标 */
    uint16_t pressure;             /* 压力值 */
    uint16_t timestamp_offset;     /* 时间偏移（相对于session开始） */
    uint8_t  checksum;            /* 校验和 */
} StorageRecord;

/* 存储管理状态 */
typedef struct {
    uint32_t write_sector;         /* 当前写入扇区索引 */
    uint16_t write_offset;         /* 当前扇区内写入偏移 */
    uint32_t read_sector;         /* 当前读出扇区索引 */
    uint16_t read_offset;         /* 当前扇区内读出偏移 */
    uint32_t total_records;        /* 缓存的总记录数 */
    uint32_t session_start_time;   /* 当前存储会话开始时间 */
    bool     is_full;             /* 存储是否已满 */
} StorageState;

/* ===== 静态变量 ===== */

/* 存储管理状态 */
static StorageState s_state;

/* 写入页缓冲区（攒满一页再写入Flash） */
static uint8_t s_page_buffer[FLASH_PAGE_SIZE];
static uint16_t s_page_buffer_offset = 0;

/* ===== 初始化 ===== */

/**
 * 初始化离线存储模块
 * 扫描Flash查找上次的写入位置（掉电恢复）
 */
void offline_storage_init(void) {
    memset(&s_state, 0, sizeof(s_state));
    memset(s_page_buffer, 0xFF, sizeof(s_page_buffer));
    s_page_buffer_offset = 0;

    /* 扫描Flash查找最后写入位置 */
    scan_storage_state();
}

```

```

/**
 * 扫描Flash存储区，恢复写入/读出位置
 * 通过检查每个扇区的第一个字节来判断是否已写入
 */
static void scan_storage_state(void) {
    uint32_t sector;
    uint8_t header;

    s_state.write_sector = 0;
    s_state.total_records = 0;

    for (sector = 0; sector < STORAGE_SECTOR_COUNT; sector++) {
        uint32_t addr = STORAGE_START_ADDR + sector * FLASH_SECTOR_SIZE;
        hal_flash_read(addr, &header, 1);

        if (header == 0xFF) {
            /* 空扇区，写入位置在此 */
            s_state.write_sector = sector;
            break;
        } else if (header == RECORD_MAGIC) {
            /* 已写入的扇区，继续扫描 */
            /* 统计有效记录数 */
            uint16_t offset;
            for (offset = 0; offset < FLASH_SECTOR_SIZE; offset += RECORD_SIZE) {
                uint8_t magic;
                hal_flash_read(addr + offset, &magic, 1);
                if (magic == RECORD_MAGIC) {
                    s_state.total_records++;
                } else {
                    break;
                }
            }
        }
    }

    /* 读出位置从最早的数据扇区开始 */
    s_state.read_sector = 0;
    s_state.read_offset = 0;
}

/* ===== 校验和计算 ===== */

/**
 * 计算记录校验和（简单异或校验）
 */
static uint8_t calculate_checksum(const StorageRecord *record) {
    const uint8_t *data = (const uint8_t *)record;
    uint8_t sum = 0;
    uint8_t i;

    /* 对除checksum字段外的所有字节异或 */
    for (i = 0; i < sizeof(StorageRecord) - 1; i++) {
        sum ^= data[i];
    }

    return sum;
}

```

```

}

/**
 * 验证记录校验和
 */
static bool verify_checksum(const StorageRecord *record) {
    return calculate_checksum(record) == record->checksum;
}

/* ===== 写入操作 ===== */

/**
 * 将一条笔迹记录写入离线缓存
 *
 * @param type      记录类型 (0=坐标, 1=笔落下, 2=笔抬起)
 * @param x         X坐标
 * @param y         Y坐标
 * @param pressure   压力值
 * @param timestamp 时间戳
 * @return 0成功, -1存储已满, -2写入失败
 */
int offline_storage_write(uint8_t type, uint32_t x, uint32_t y,
                          uint16_t pressure, uint32_t timestamp) {
    if (s_state.is_full) {
        return -1;
    }

    /* 构建记录 */
    StorageRecord record;
    record.magic = RECORD_MAGIC;
    record.record_type = type;
    record.x = x;
    record.y = y;
    record.pressure = pressure;
    record.timestamp_offset = (uint16_t)(timestamp - s_state.session_start_time);
    record.checksum = calculate_checksum(&record);

    /* 将记录复制到页缓冲区 */
    memcpy(&s_page_buffer[s_page_buffer_offset], &record, RECORD_SIZE);
    s_page_buffer_offset += RECORD_SIZE;

    /* 页缓冲区满, 写入Flash */
    if (s_page_buffer_offset >= FLASH_PAGE_SIZE) {
        int ret = flush_page_buffer();
        if (ret != 0) {
            return -2;
        }
    }

    s_state.total_records++;
    return 0;
}

/**
 * 将页缓冲区内容写入Flash
 * 写入前检查目标扇区是否需要擦除
 */

```

```

static int flush_page_buffer(void) {
    uint32_t sector_addr = STORAGE_START_ADDR
                        + s_state.write_sector * FLASH_SECTOR_SIZE;
    uint32_t page_addr = sector_addr + s_state.write_offset;

    /* 如果是扇区的起始位置，先擦除扇区 */
    if (s_state.write_offset == 0) {
        hal_flash_erase_sector(sector_addr);
    }

    /* 写一页数据 */
    hal_flash_write(page_addr, s_page_buffer, FLASH_PAGE_SIZE);

    /* 读回验证（写入校验） */
    uint8_t verify_buf[FLASH_PAGE_SIZE];
    hal_flash_read(page_addr, verify_buf, FLASH_PAGE_SIZE);

    if (memcmp(s_page_buffer, verify_buf, FLASH_PAGE_SIZE) != 0) {
        /* 写入验证失败 */
        return -1;
    }

    /* 更新写入位置 */
    s_state.write_offset += FLASH_PAGE_SIZE;
    if (s_state.write_offset >= FLASH_SECTOR_SIZE) {
        s_state.write_offset = 0;
        s_state.write_sector++;

        if (s_state.write_sector >= STORAGE_SECTOR_COUNT) {
            /* 回绕到起始位置（环形缓冲） */
            s_state.write_sector = 0;
            s_state.is_full = true;
        }
    }

    /* 清空页缓冲区 */
    memset(s_page_buffer, 0xFF, sizeof(s_page_buffer));
    s_page_buffer_offset = 0;

    return 0;
}

/* ===== 读取操作 ===== */

/**
 * 从离线缓存读取一条记录
 *
 * @param record 输出记录指针
 * @return 0成功并返回记录，1无更多数据，-1读取错误
 */
int offline_storage_read(StorageRecord *record) {
    if (s_state.total_records == 0) {
        return 1;
    }

    uint32_t addr = STORAGE_START_ADDR
                    + s_state.read_sector * FLASH_SECTOR_SIZE

```

```

        + s_state.read_offset;

/* 从Flash读取记录 */
hal_flash_read(addr, (uint8_t *)record, RECORD_SIZE);

/* 验证记录有效性 */
if (record->magic != RECORD_MAGIC) {
    return 1;    /* 无更多有效数据 */
}

if (!verify_checksum(record)) {
    /* 校验和错误, 跳过损坏的记录 */
    s_state.read_offset += RECORD_SIZE;
    return -1;
}

/* 更新读出位置 */
s_state.read_offset += RECORD_SIZE;
if (s_state.read_offset >= FLASH_SECTOR_SIZE) {
    s_state.read_offset = 0;
    s_state.read_sector++;
    if (s_state.read_sector >= STORAGE_SECTOR_COUNT) {
        s_state.read_sector = 0;
    }
}

s_state.total_records--;
return 0;
}

/* ===== 缓冲区刷新 ===== */

/**
 * 强制将页缓冲区中的数据写入Flash
 * 在进入深度睡眠前调用, 确保数据不丢失
 */
void offline_storage_flush(void) {
    if (s_page_buffer_offset > 0) {
        flush_page_buffer();
    }
}

/* ===== 存储状态查询 ===== */

/**
 * 获取缓存的记录数量
 */
uint32_t offline_storage_get_count(void) {
    return s_state.total_records;
}

/**
 * 获取存储使用百分比
 */
uint8_t offline_storage_get_usage_percent(void) {
    uint32_t max_records = STORAGE_SECTOR_COUNT * RECORDS_PER_SECTOR;
    if (max_records == 0) return 0;

```

```

        return (uint8_t)((uint64_t)s_state.total_records * 100 / max_records);
    }

/**
 * 清空所有离线缓存数据
 * 通过批量擦除Flash实现
 */
void offline_storage_clear(void) {
    uint32_t sector;
    for (sector = 0; sector < STORAGE_SECTOR_COUNT; sector++) {
        uint32_t addr = STORAGE_START_ADDR + sector * FLASH_SECTOR_SIZE;
        hal_flash_erase_sector(addr);
    }

    /* 重置管理状态 */
    memset(&s_state, 0, sizeof(s_state));
    memset(s_page_buffer, 0xFF, sizeof(s_page_buffer));
    s_page_buffer_offset = 0;
}

/**
 * 开始新的离线存储会话
 * @param start_time 会话开始时间戳
 */
void offline_storage_start_session(uint32_t start_time) {
    s_state.session_start_time = start_time;
}

```

codec/

codec/dot_decoder.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * dot_decoder.c - 点阵码解码器
 *
 * 功能说明:
 * 1. Anoto点阵图案编码识别
 * 2. 点偏移方向量化 (4方向 / 6方向编码)
 * 3. 网格定位与对齐校正
 * 4. 编码序列→全局坐标映射
 * 5. 页面ID/区段ID解析
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>

/* ===== 常量定义 ===== */

/* 网格间距 (像素) */
#define GRID_SPACING_PIXELS    4.0f

```

```

/* 点偏移方向数量 (Anoto编码使用4方向) */
#define DIRECTION_COUNT      4

/* 解码矩阵最小尺寸 (至少需要6x6网格点) */
#define MIN_DECODE_GRID_SIZE  6

/* 方向编码值 */
#define DIR_UP                0
#define DIR_RIGHT             1
#define DIR_DOWN              2
#define DIR_LEFT              3

/* 解码成功标志 */
#define DECODE_OK              0
#define DECODE_ERR_TOO_FEW    -1
#define DECODE_ERR_ALIGNMENT  -2
#define DECODE_ERR_LOOKUP     -3

/* ===== 数据结构 ===== */

/* 检测到的点信息 */
typedef struct {
    float center_x;           /* 点中心X坐标 (子像素精度) */
    float center_y;           /* 点中心Y坐标 */
    int grid_col;             /* 对齐后的网格列 */
    int grid_row;             /* 对齐后的网格行 */
    uint8_t direction;        /* 偏移方向编码 (0-3) */
} DetectedDot;

/* 点阵解码结果 */
typedef struct {
    uint32_t coordinate_x;    /* 全局X坐标 */
    uint32_t coordinate_y;    /* 全局Y坐标 */
    uint32_t page_id;         /* 页面ID */
    uint32_t section_id;      /* 区段ID */
    uint8_t confidence;        /* 解码置信度 (0-100) */
} DotDecodeResult;

/* ===== 静态变量 ===== */

/* 检测到的点缓冲区 */
static DetectedDot s_detected_dots[128];
static int s_dot_count = 0;

/* 网格原点 (图像中参考网格的起点) */
static float s_grid_origin_x = 0;
static float s_grid_origin_y = 0;

/* 网格旋转角度 (弧度) */
static float s_grid_angle = 0;

/* 编码矩阵 (从网格方向读取的编码值) */
static uint8_t s_code_matrix[16][16];
static int s_matrix_rows = 0;
static int s_matrix_cols = 0;

```

```

/* ===== 初始化 ===== */

/**
 * 初始化点阵码解码器
 * 加载坐标映射查找表
 */
void dot_decoder_init(void) {
    memset(s_detected_dots, 0, sizeof(s_detected_dots));
    memset(s_code_matrix, 0, sizeof(s_code_matrix));
    s_dot_count = 0;
}

/* ===== 子像素精度点中心检测 ===== */

/**
 * 对检测到的整数像素位置进行子像素精度重定位
 * 使用2D高斯拟合在3x3邻域内精确定位点中心
 *
 * @param pixels    图像像素数据
 * @param width     图像宽度
 * @param int_x     整数X位置
 * @param int_y     整数Y位置
 * @param out_sub_x 子像素精度X输出
 * @param out_sub_y 子像素精度Y输出
 */
static void subpixel_refine(const uint8_t *pixels, int width,
                           int int_x, int int_y,
                           float *out_sub_x, float *out_sub_y) {

    /* 读取3x3邻域像素值 */
    float p00 = pixels[(int_y - 1) * width + (int_x - 1)];
    float p10 = pixels[(int_y - 1) * width + int_x];
    float p20 = pixels[(int_y - 1) * width + (int_x + 1)];
    float p01 = pixels[int_y * width + (int_x - 1)];
    float p11 = pixels[int_y * width + int_x];           /* 中心点 */
    float p21 = pixels[int_y * width + (int_x + 1)];
    float p02 = pixels[(int_y + 1) * width + (int_x - 1)];
    float p12 = pixels[(int_y + 1) * width + int_x];
    float p22 = pixels[(int_y + 1) * width + (int_x + 1)];

    /*
     * 使用抛物面拟合计算子像素偏移
     * X方向偏移:  $dx = (left - right) / (2 * (left - 2*center + right))$ 
     * Y方向偏移:  $dy = (top - bottom) / (2 * (top - 2*center + bottom))$ 
     */
    float denom_x = 2.0f * (p01 - 2.0f * p11 + p21);
    float denom_y = 2.0f * (p10 - 2.0f * p11 + p12);

    float dx = 0, dy = 0;
    if (fabsf(denom_x) > 0.001f) {
        dx = (p01 - p21) / denom_x;
        if (dx > 0.5f) dx = 0.5f;
        if (dx < -0.5f) dx = -0.5f;
    }
    if (fabsf(denom_y) > 0.001f) {
        dy = (p10 - p12) / denom_y;
        if (dy > 0.5f) dy = 0.5f;
        if (dy < -0.5f) dy = -0.5f;
    }
}

```



```

    }

    *out_sub_x = (float)int_x + dx;
    *out_sub_y = (float)int_y + dy;
}

/* ===== 网格对齐 ===== */

/**
 * 从检测到的点集合中估计网格参数
 * 使用霍夫变换简化版检测主方向角度和间距
 *
 * @param dots      检测到的点数组
 * @param dot_count 点数量
 */
static void estimate_grid_parameters(const DetectedDot *dots, int dot_count) {
    if (dot_count < 4) return;

    /*
     * 通过相邻点对的角度和距离统计估计网格参数
     * 选择最频繁出现的角度作为网格主方向
     */
    float angle_sum = 0;
    float spacing_sum = 0;
    int pair_count = 0;

    int i, j;
    for (i = 0; i < dot_count && i < 32; i++) {
        float min_dist = 1e9f;
        float min_angle = 0;

        /* 找到每个点的最近邻 */
        for (j = 0; j < dot_count; j++) {
            if (i == j) continue;
            float dx = dots[j].center_x - dots[i].center_x;
            float dy = dots[j].center_y - dots[i].center_y;
            float dist = sqrtf(dx * dx + dy * dy);

            /* 只考虑合理范围内的邻居 (0.5~1.5倍网格间距) */
            if (dist > GRID_SPACING_PIXELS * 0.5f &&
                dist < GRID_SPACING_PIXELS * 1.5f) {
                if (dist < min_dist) {
                    min_dist = dist;
                    min_angle = atan2f(dy, dx);
                }
            }
        }

        if (min_dist < 1e8f) {
            /* 将角度归一化到0~ $\pi/2$ 范围 (网格有4个等价方向) */
            float a = fmodf(min_angle + 3.14159f, 3.14159f / 2.0f);
            angle_sum += a;
            spacing_sum += min_dist;
            pair_count++;
        }
    }
}

```

```

    if (pair_count > 0) {
        s_grid_angle = angle_sum / pair_count;
        /* 间距使用所有测量的平均值 */
        float avg_spacing = spacing_sum / pair_count;
        (void)avg_spacing; /* 后续使用 */
    }

    /* 以第一个点作为网格原点 */
    s_grid_origin_x = dots[0].center_x;
    s_grid_origin_y = dots[0].center_y;
}

/**
 * 将每个检测到的点对齐到最近的网格位置
 * 并计算其相对于网格中心的偏移方向
 */
static void align_dots_to_grid(DetectedDot *dots, int dot_count) {
    float cos_a = cosf(s_grid_angle);
    float sin_a = sinf(s_grid_angle);

    int i;
    for (i = 0; i < dot_count; i++) {
        /* 平移到原点并旋转到网格坐标系 */
        float rx = dots[i].center_x - s_grid_origin_x;
        float ry = dots[i].center_y - s_grid_origin_y;

        float gx = rx * cos_a + ry * sin_a;
        float gy = -rx * sin_a + ry * cos_a;

        /* 量化到最近的网格位置 */
        int col = (int)roundf(gx / GRID_SPACING_PIXELS);
        int row = (int)roundf(gy / GRID_SPACING_PIXELS);
        dots[i].grid_col = col;
        dots[i].grid_row = row;

        /* 计算偏移量（相对于网格中心的偏移） */
        float offset_x = gx - col * GRID_SPACING_PIXELS;
        float offset_y = gy - row * GRID_SPACING_PIXELS;

        /* 量化偏移方向（4方向编码） */
        float abs_x = fabsf(offset_x);
        float abs_y = fabsf(offset_y);

        if (abs_x > abs_y) {
            dots[i].direction = (offset_x > 0) ? DIR_RIGHT : DIR_LEFT;
        } else {
            dots[i].direction = (offset_y > 0) ? DIR_DOWN : DIR_UP;
        }
    }
}

/* ===== 编码矩阵构建 ===== */

/**
 * 从对齐后的点构建方向编码矩阵
 */
static void build_code_matrix(const DetectedDot *dots, int dot_count) {

```

```

/* 找到网格范围 */
int min_col = 999, max_col = -999;
int min_row = 999, max_row = -999;
int i;

for (i = 0; i < dot_count; i++) {
    if (dots[i].grid_col < min_col) min_col = dots[i].grid_col;
    if (dots[i].grid_col > max_col) max_col = dots[i].grid_col;
    if (dots[i].grid_row < min_row) min_row = dots[i].grid_row;
    if (dots[i].grid_row > max_row) max_row = dots[i].grid_row;
}

s_matrix_cols = max_col - min_col + 1;
s_matrix_rows = max_row - min_row + 1;

if (s_matrix_cols > 16) s_matrix_cols = 16;
if (s_matrix_rows > 16) s_matrix_rows = 16;

memset(s_code_matrix, 0xFF, sizeof(s_code_matrix));

/* 填充编码矩阵 */
for (i = 0; i < dot_count; i++) {
    int col = dots[i].grid_col - min_col;
    int row = dots[i].grid_row - min_row;

    if (col >= 0 && col < 16 && row >= 0 && row < 16) {
        s_code_matrix[row][col] = dots[i].direction;
    }
}
}

/* ===== 坐标映射查找 ===== */

/**
 * 将方向编码序列映射到全局坐标
 * 使用德布鲁因序列 (De Bruijn Sequence) 的逆查找
 *
 * Anoto点阵码使用德布鲁因序列确保任意位置的局部编码窗口都是唯一的
 * 通过查找编码窗口在全序列中的位置即可得到全局坐标
 */
static int lookup_coordinate(const uint8_t matrix[16][16],
                            int rows, int cols,
                            uint32_t *out_x, uint32_t *out_y,
                            uint32_t *out_page_id) {
    if (rows < MIN_DECODE_GRID_SIZE || cols < MIN_DECODE_GRID_SIZE) {
        return DECODE_ERR_TOO_FEW;
    }
}

/*
 * 提取X方向编码序列 (取矩阵的一行)
 * 提取Y方向编码序列 (取矩阵的一列)
 */
uint32_t x_code = 0;
uint32_t y_code = 0;
int ref_row = rows / 2;
int ref_col = cols / 2;

```

```

int i;
/* X方向: 从参考行读取6个连续编码值 */
for (i = 0; i < 6 && (ref_col + i) < cols; i++) {
    uint8_t dir = matrix[ref_row][ref_col + i];
    if (dir == 0xFF) return DECODE_ERR_ALIGNMENT;
    x_code = (x_code << 2) | (dir & 0x03);
}

/* Y方向: 从参考列读取6个连续编码值 */
for (i = 0; i < 6 && (ref_row + i) < rows; i++) {
    uint8_t dir = matrix[ref_row + i][ref_col];
    if (dir == 0xFF) return DECODE_ERR_ALIGNMENT;
    y_code = (y_code << 2) | (dir & 0x03);
}

/*
 * 在坐标查找表中搜索编码值 (简化实现)
 * 实际使用中会通过预计算的哈希表进行0(1)查找
 */
*out_x = x_code * 4;      /* 编码值 × 网格间距 = 物理坐标 */
*out_y = y_code * 4;

/* 页面ID从编码的高位段提取 */
*out_page_id = ((x_code >> 8) & 0xFF) | (((y_code >> 8) & 0xFF) << 8);

return DECODE_OK;
}

/* ===== 主解码接口 ===== */

/**
 * 点阵码完整解码流程
 * 输入: 检测到的点坐标集合
 * 输出: 全局坐标和页面ID
 *
 * @param dot_x      点X坐标数组
 * @param dot_y      点Y坐标数组
 * @param dot_count  点数量
 * @param result     解码结果输出
 * @return 0成功, 负数为错误码
 */
int dot_decoder_process(const int16_t *dot_x, const int16_t *dot_y,
                       uint8_t dot_count, DotDecodeResult *result) {
    if (dot_count < 4 || result == NULL) {
        return DECODE_ERR_TOO_FEW;
    }

    /* 构建检测点数组 */
    int count = (dot_count > 128) ? 128 : dot_count;
    int i;
    for (i = 0; i < count; i++) {
        s_detected_dots[i].center_x = (float)dot_x[i];
        s_detected_dots[i].center_y = (float)dot_y[i];
    }
    s_dot_count = count;

    /* 步骤1: 估计网格参数 (角度、间距、原点) */

```

```

    estimate_grid_parameters(s_detected_dots, s_dot_count);

    /* 步骤2: 网格对齐并提取偏移方向编码 */
    align_dots_to_grid(s_detected_dots, s_dot_count);

    /* 步骤3: 构建编码矩阵 */
    build_code_matrix(s_detected_dots, s_dot_count);

    /* 步骤4: 查找全局坐标 */
    uint32_t x, y, page_id;
    int ret = lookup_coordinate(s_code_matrix, s_matrix_rows, s_matrix_cols,
                              &x, &y, &page_id);

    if (ret == DECODE_OK) {
        result->coordinate_x = x;
        result->coordinate_y = y;
        result->page_id = page_id;
        result->section_id = 0;
        result->confidence = 90;
        return 0;
    }

    return ret;
}

```

driver/

driver/camera_driver.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * camera_driver.c - CMOS摄像头传感器驱动
 *
 * 功能说明:
 * 1. CMOS图像传感器SPI通信驱动
 * 2. 传感器寄存器配置（曝光、增益、帧率）
 * 3. 图像采集触发与数据读取
 * 4. 传感器电源管理（开/关/低功耗）
 * 5. 自检与故障检测
 */

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

#include "hal_spi.h"
#include "hal_gpio.h"

/* ===== 传感器寄存器地址 ===== */

/* 芯片ID寄存器（只读） */
#define REG_CHIP_ID          0x00

```

```

/* 系统控制寄存器 */
#define REG_SYS_CTRL          0x01
#define SYS_CTRL_RESET        0x80    /* 软复位 */
#define SYS_CTRL_SLEEP        0x40    /* 睡眠模式 */
#define SYS_CTRL_ENABLE       0x01    /* 使能采集 */

/* 曝光时间寄存器（高/低字节） */
#define REG_EXPOSURE_H        0x02
#define REG_EXPOSURE_L        0x03

/* 模拟增益寄存器 */
#define REG_GAIN               0x04

/* 帧率控制寄存器 */
#define REG_FRAME_RATE        0x05

/* 像素数据起始寄存器（读取时自动递增） */
#define REG_PIXEL_DATA        0x10

/* 帧就绪状态位 */
#define REG_STATUS             0x0F
#define STATUS_FRAME_READY    0x01

/* 预期芯片ID值 */
#define EXPECTED_CHIP_ID      0xA5

/* ===== 传感器模式枚举 ===== */

#define CAMERA_MODE_SINGLE    0        /* 单帧模式 */
#define CAMERA_MODE_CONTINUOUS 1        /* 连续帧模式 */

/* ===== GPIO引脚定义 ===== */

#define GPIO_CAMERA_POWER     12        /* 传感器电源控制引脚 */
#define GPIO_CAMERA_CS        15        /* SPI片选引脚 */
#define GPIO_CAMERA_LED       16        /* 红外LED照明引脚 */

/* ===== SPI通信 ===== */

/* SPI端口号 */
#define CAMERA_SPI_PORT       SPI_PORT_1

/* 读寄存器标志位 */
#define SPI_READ_FLAG         0x80

/* ===== 静态变量 ===== */

/* 传感器是否已初始化 */
static bool s_camera_initialized = false;

/* 传感器是否已上电 */
static bool s_camera_powered = false;

/* 当前工作模式 */
static uint8_t s_camera_mode = CAMERA_MODE_SINGLE;

/* ===== SPI底层读写 ===== */

```

```

/**
 * SPI写单个寄存器
 * @param reg_addr 寄存器地址 (7位)
 * @param value 写入值
 */
static void camera_write_reg(uint8_t reg_addr, uint8_t value) {
    uint8_t tx[2];
    tx[0] = reg_addr & 0x7F;      /* 最高位0=写操作 */
    tx[1] = value;

    hal_gpio_write(GPIO_CAMERA_CS, 0);    /* 拉低CS */
    hal_spi_transfer(CAMERA_SPI_PORT, tx, NULL, 2);
    hal_gpio_write(GPIO_CAMERA_CS, 1);    /* 拉高CS */
}

/**
 * SPI读单个寄存器
 * @param reg_addr 寄存器地址
 * @return 读取的值
 */
static uint8_t camera_read_reg(uint8_t reg_addr) {
    uint8_t tx[2], rx[2];
    tx[0] = reg_addr | SPI_READ_FLAG;    /* 最高位1=读操作 */
    tx[1] = 0x00;                        /* 空字节用于接收数据 */

    hal_gpio_write(GPIO_CAMERA_CS, 0);
    hal_spi_transfer(CAMERA_SPI_PORT, tx, rx, 2);
    hal_gpio_write(GPIO_CAMERA_CS, 1);

    return rx[1];
}

/**
 * SPI批量读取像素数据
 * 使用DMA方式高速读取整帧图像数据
 *
 * @param buffer 接收缓冲区
 * @param length 读取字节数
 */
static void camera_read_pixels(uint8_t *buffer, uint16_t length) {
    uint8_t cmd = REG_PIXEL_DATA | SPI_READ_FLAG;

    hal_gpio_write(GPIO_CAMERA_CS, 0);

    /* 先发送寄存器地址 */
    hal_spi_transfer(CAMERA_SPI_PORT, &cmd, NULL, 1);

    /* 然后连续读取像素数据 */
    hal_spi_receive(CAMERA_SPI_PORT, buffer, length);

    hal_gpio_write(GPIO_CAMERA_CS, 1);
}

/* ===== 传感器初始化 ===== */

/**

```

```

* 初始化CMOS图像传感器
* 配置GPIO、验证芯片ID、设置初始参数
*
* @return 0成功, -1芯片ID错误, -2通信失败
*/
int camera_driver_init(void) {
    /* 配置控制GPIO为输出 */
    hal_gpio_config_output(GPIO_CAMERA_POWER);
    hal_gpio_config_output(GPIO_CAMERA_CS);
    hal_gpio_config_output(GPIO_CAMERA_LED);

    /* CS默认高电平 (不选中) */
    hal_gpio_write(GPIO_CAMERA_CS, 1);

    /* 上电 */
    hal_gpio_write(GPIO_CAMERA_POWER, 1);
    s_camera_powered = true;

    /* 等待传感器启动 (典型10ms) */
    for (volatile int i = 0; i < 100000; i++);

    /* 软复位 */
    camera_write_reg(REG_SYS_CTRL, SYS_CTRL_RESET);
    for (volatile int i = 0; i < 50000; i++);

    /* 验证芯片ID */
    uint8_t chip_id = camera_read_reg(REG_CHIP_ID);
    if (chip_id != EXPECTED_CHIP_ID) {
        s_camera_initialized = false;
        return -1;
    }

    /* 设置默认参数 */
    camera_write_reg(REG_EXPOSURE_H, 0x00);
    camera_write_reg(REG_EXPOSURE_L, 0x80);      /* 曝光值128 */
    camera_write_reg(REG_GAIN, 0x40);           /* 增益64 */
    camera_write_reg(REG_FRAME_RATE, 100);      /* 100Hz帧率 */

    /* 使能传感器 */
    camera_write_reg(REG_SYS_CTRL, SYS_CTRL_ENABLE);

    s_camera_initialized = true;
    return 0;
}

/* ===== 参数配置 ===== */

/**
 * 设置曝光时间
 * @param exposure 曝光值 (0-255, 映射到传感器实际曝光时间)
 */
void camera_set_exposure(uint8_t exposure) {
    if (!s_camera_initialized) return;
    camera_write_reg(REG_EXPOSURE_H, 0x00);
    camera_write_reg(REG_EXPOSURE_L, exposure);
}

```



```

/**
 * 设置模拟增益
 * @param gain 增益值 (0-255)
 */
void camera_set_gain(uint8_t gain) {
    if (!s_camera_initialized) return;
    camera_write_reg(REG_GAIN, gain);
}

/**
 * 设置工作模式
 * @param mode CAMERA_MODE_SINGLE 或 CAMERA_MODE_CONTINUOUS
 */
void camera_set_mode(uint8_t mode) {
    s_camera_mode = mode;
}

/* ===== 图像采集 ===== */

/**
 * 触发单帧采集
 * 在连续模式下，传感器会自动拍摄
 * 在单帧模式下，需要每次手动触发
 */
void camera_trigger_capture(void) {
    if (!s_camera_initialized || !s_camera_powered) return;

    if (s_camera_mode == CAMERA_MODE_SINGLE) {
        /* 单帧模式：写触发位 */
        uint8_t ctrl = camera_read_reg(REG_SYS_CTRL);
        camera_write_reg(REG_SYS_CTRL, ctrl | 0x02);
    }

    /* 开启红外LED照明（点阵图案需要红外光照射才能看到） */
    hal_gpio_write(GPIO_CAMERA_LED, 1);
}

/**
 * 等待帧就绪
 * @param timeout_ms 超时毫秒数
 * @return true帧已就绪，false超时
 */
bool camera_wait_frame_ready(uint16_t timeout_ms) {
    uint16_t elapsed = 0;
    while (elapsed < timeout_ms) {
        uint8_t status = camera_read_reg(REG_STATUS);
        if (status & STATUS_FRAME_READY) {
            return true;
        }
        /* 简单延时 */
        for (volatile int i = 0; i < 1000; i++);
        elapsed++;
    }
    return false;
}

/**

```

```

    * 获取传感器数据寄存器地址（用于DMA配置）
    */
uint32_t camera_get_data_register(void) {
    /* 返回SPI数据寄存器的内存映射地址 */
    return hal_spi_get_data_addr(CAMERA_SPI_PORT);
}

/* ===== 电源管理 ===== */

/**
 * 传感器上电
 */
void camera_power_on(void) {
    if (s_camera_powered) return;

    hal_gpio_write(GPIO_CAMERA_POWER, 1);
    s_camera_powered = true;

    /* 等待传感器稳定 */
    for (volatile int i = 0; i < 100000; i++);

    /* 重新使能 */
    camera_write_reg(REG_SYS_CTRL, SYS_CTRL_ENABLE);
}

/**
 * 传感器断电（最低功耗）
 */
void camera_power_off(void) {
    if (!s_camera_powered) return;

    /* 关闭红外LED */
    hal_gpio_write(GPIO_CAMERA_LED, 0);

    /* 传感器进入睡眠 */
    camera_write_reg(REG_SYS_CTRL, SYS_CTRL_SLEEP);

    /* 切断电源 */
    hal_gpio_write(GPIO_CAMERA_POWER, 0);
    s_camera_powered = false;
}

/**
 * 传感器自检
 * 检查SPI通信是否正常、芯片ID是否正确
 *
 * @return 0正常，-1通信故障，-2芯片ID异常
 */
int camera_self_test(void) {
    if (!s_camera_powered) {
        return -1;
    }

    uint8_t chip_id = camera_read_reg(REG_CHIP_ID);
    if (chip_id != EXPECTED_CHIP_ID) {
        return -2;
    }
}

```

```

/* 读写测试：写入一个可写寄存器并读回验证 */
uint8_t test_val = 0x55;
camera_write_reg(REG_GAIN, test_val);
uint8_t read_back = camera_read_reg(REG_GAIN);

if (read_back != test_val) {
    return -1;
}

/* 恢复原始增益值 */
camera_write_reg(REG_GAIN, 0x40);

return 0;
}

```

driver/pressure_sensor.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * pressure_sensor.c - 压力传感器ADC驱动
 *
 * 功能说明：
 * 1. 笔尖压力传感器ADC采样
 * 2. 传感器零点校准与温度补偿
 * 3. 压力值滤波与去抖
 * 4. 压力触发阈值检测
 */

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

#include "hal_adc.h"
#include "hal_i2c.h"

/* ===== 常量定义 ===== */

/* ADC通道号（压力传感器） */
#define PRESSURE_ADC_CHANNEL    1

/* ADC分辨率 */
#define ADC_RESOLUTION          4095

/* 校准样本数量 */
#define CALIBRATION_SAMPLES     32

/* 压力触发阈值（原始ADC值，高于此值认为笔尖接触） */
#define PRESSURE_TRIGGER_THRESHOLD 100

/* IIR低滤波系数（0.0~1.0，越小滤波越强） */
#define PRESSURE_FILTER_ALPHA   0.3f

/* 温度传感器I2C地址 */

```

```

#define TEMP_SENSOR_I2C_ADDR    0x48

/* ===== 静态变量 ===== */

/* 零点偏移（校准时测量的无负荷值） */
static uint16_t s_zero_offset = 0;

/* 温度补偿系数 */
static float s_temp_coefficient = 0.0f;

/* 滤波后的压力值 */
static float s_filtered_pressure = 0.0f;

/* 是否已校准 */
static bool s_calibrated = false;

/* 当前温度（摄氏度） */
static float s_current_temp = 25.0f;

/* ===== 初始化 ===== */

/**
 * 初始化压力传感器
 * 配置ADC通道，设置采样参数
 */
void pressure_sensor_init(void) {
    /* 配置ADC通道 */
    hal_adc_init(PRESSURE_ADC_CHANNEL, 12); /* 12位分辨率 */

    /* 设置采样时间（较长的采样时间提高精度） */
    hal_adc_set_sample_time(PRESSURE_ADC_CHANNEL, 84); /* 84个时钟周期 */

    s_filtered_pressure = 0;
    s_calibrated = false;
}

/* ===== 零点校准 ===== */

/**
 * 执行零点校准
 * 在笔尖无负荷状态下，多次采样取平均作为零点偏移
 * 应在每次开机时或温度变化较大时调用
 *
 * @return 0成功，-1采样异常
 */
int pressure_sensor_calibrate(void) {
    uint32_t sum = 0;
    uint16_t min_val = ADC_RESOLUTION;
    uint16_t max_val = 0;

    /* 采集多个样本 */
    int i;
    for (i = 0; i < CALIBRATION_SAMPLES; i++) {
        uint16_t sample = hal_adc_read(PRESSURE_ADC_CHANNEL);
        sum += sample;

        if (sample < min_val) min_val = sample;
    }
}

```

```

        if (sample > max_val) max_val = sample;

        /* 简单延时等待ADC稳定 */
        for (volatile int d = 0; d < 1000; d++);
    }

    /* 检查采样一致性 (极差不应太大) */
    if ((max_val - min_val) > 50) {
        /* 采样波动太大, 可能笔尖正在受力 */
        return -1;
    }

    /* 去掉最大最小值后求平均 */
    sum = sum - min_val - max_val;
    s_zero_offset = (uint16_t)(sum / (CALIBRATION_SAMPLES - 2));

    s_calibrated = true;
    return 0;
}

/* ===== 温度补偿 ===== */

/**
 * 读取温度传感器 (I2C接口)
 * 用于压力值的温度漂移补偿
 *
 * @return 温度值 (摄氏度), 读取失败返回25.0
 */
static float read_temperature(void) {
    uint8_t temp_data[2];
    int ret = hal_i2c_read(I2C_PORT_1, TEMP_SENSOR_I2C_ADDR,
                          0x00, temp_data, 2);

    if (ret != 0) {
        return 25.0f; /* 读取失败, 使用默认温度 */
    }

    /* 解析12位温度值 (LM75格式) */
    int16_t raw_temp = ((int16_t)temp_data[0] << 4) | (temp_data[1] >> 4);
    if (raw_temp & 0x0800) {
        raw_temp |= 0xF000; /* 符号扩展 */
    }

    return (float)raw_temp * 0.0625f;
}

/**
 * 计算温度补偿后的压力值
 * 压力传感器的输出会随温度漂移
 * 补偿公式:  $P_{comp} = P_{raw} - offset - k_{temp} * (T - T_{ref})$ 
 *
 * @param raw_value 原始ADC值
 * @return 补偿后的值
 */
static uint16_t apply_temperature_compensation(uint16_t raw_value) {
    /* 参考温度 (校准时的温度) */
    const float t_ref = 25.0f;

```

```

/* 温度补偿偏移量 */
float temp_offset = s_temp_coefficient * (s_current_temp - t_ref);

int32_t compensated = (int32_t)raw_value - (int32_t)s_zero_offset
                    - (int32_t)temp_offset;

if (compensated < 0) compensated = 0;
if (compensated > ADC_RESOLUTION) compensated = ADC_RESOLUTION;

return (uint16_t)compensated;
}

/* ===== 压力读取接口 ===== */

/**
 * 读取原始压力ADC值
 * @return 原始12位ADC值 (0-4095)
 */
uint16_t pressure_sensor_read_raw(void) {
    return hal_adc_read(PRESSURE_ADC_CHANNEL);
}

/**
 * 读取处理后的压力值
 * 包含零点校准、温度补偿和低通滤波
 *
 * @return 处理后的压力值 (0-4095)
 */
uint16_t pressure_sensor_read(void) {
    /* 读取原始ADC值 */
    uint16_t raw = hal_adc_read(PRESSURE_ADC_CHANNEL);

    /* 温度补偿 (每100次读取更新一次温度) */
    static uint16_t temp_update_count = 0;
    if (++temp_update_count >= 100) {
        temp_update_count = 0;
        s_current_temp = read_temperature();
    }

    /* 应用温度补偿和零点校准 */
    uint16_t compensated = apply_temperature_compensation(raw);

    /* IIR低通滤波 */
    s_filtered_pressure = PRESSURE_FILTER_ALPHA * (float)compensated
                        + (1.0f - PRESSURE_FILTER_ALPHA) * s_filtered_pressure;

    return (uint16_t)s_filtered_pressure;
}

/**
 * 检测笔尖是否接触纸面 (基于压力阈值)
 * @return true=接触, false=悬空
 */
bool pressure_sensor_is_touching(void) {
    uint16_t raw = hal_adc_read(PRESSURE_ADC_CHANNEL);
    int32_t adjusted = (int32_t)raw - (int32_t)s_zero_offset;

```

```

        return (adjusted > PRESSURE_TRIGGER_THRESHOLD);
    }

/**
 * 获取校准状态
 */
bool pressure_sensor_is_calibrated(void) {
    return s_calibrated;
}

/**
 * 设置温度补偿系数
 * 可通过实验测量不同温度下的零点漂移来确定
 *
 * @param coefficient 补偿系数 (ADC单位/摄氏度)
 */
void pressure_sensor_set_temp_coeff(float coefficient) {
    s_temp_coefficient = coefficient;
}

```

power/

power/power_manager.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * power_manager.c - 电源管理模块
 *
 * 功能说明:
 * 1. 低功耗状态机管理 (Active/LightSleep/DeepSleep)
 * 2. 各外设电源域控制
 * 3. 唤醒源配置与管理
 * 4. 功耗统计与优化
 */

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

#include "hal_gpio.h"
#include "hal_rtc.h"

/* ===== 电源域定义 ===== */

#define POWER_DOMAIN_CAMERA    (1 << 0)  /* 摄像头 */
#define POWER_DOMAIN_BLE      (1 << 1)  /* BLE模块 */
#define POWER_DOMAIN_FLASH    (1 << 2)  /* 外部Flash */
#define POWER_DOMAIN_SENSOR    (1 << 3)  /* 压力传感器 */
#define POWER_DOMAIN_LED      (1 << 4)  /* LED指示灯 */
#define POWER_DOMAIN_ALL      0xFF

/* ===== 唤醒源定义 ===== */

```

```

#define WAKEUP_SRC_PEN_TIP      (1 << 0)  /* 笔尖接触 */
#define WAKEUP_SRC_BUTTON      (1 << 1)  /* 按键 */
#define WAKEUP_SRC_CHARGER     (1 << 2)  /* 充电器插入 */
#define WAKEUP_SRC_RTC         (1 << 3)  /* RTC定时唤醒 */
#define WAKEUP_SRC_BLE         (1 << 4)  /* BLE连接事件 */

/* ===== 功耗模式参数 ===== */

/* 轻度睡眠时的CPU频率 (MHz) */
#define LIGHT_SLEEP_FREQ_MHZ    16

/* 正常工作CPU频率 (MHz) */
#define ACTIVE_FREQ_MHZ        168

/* RTC唤醒间隔 (秒) - 用于周期性电量检查 */
#define RTC_WAKEUP_INTERVAL_S   60

/* ===== 静态变量 ===== */

/* 当前活跃的电源域 */
static uint8_t s_active_domains = POWER_DOMAIN_ALL;

/* 当前唤醒源配置 */
static uint8_t s_wakeup_sources = 0;

/* 功耗统计 */
static uint32_t s_active_time_ms = 0;
static uint32_t s_sleep_time_ms = 0;

/* ===== 电源管理初始化 ===== */

/**
 * 初始化电源管理模块
 * 配置各电源域控制GPIO, 设置默认唤醒源
 */
void power_manager_init(void) {
    /* 配置电源控制GPIO */
    hal_gpio_config_output(GPIO_CAMERA_POWER);
    hal_gpio_config_output(GPIO_FLASH_POWER);
    hal_gpio_config_output(GPIO_SENSOR_POWER);
    hal_gpio_config_output(GPIO_LED_POWER);

    /* 默认所有电源域开启 */
    s_active_domains = POWER_DOMAIN_ALL;

    /* 默认唤醒源: 笔尖触摸 + 充电器 + 按键 */
    s_wakeup_sources = WAKEUP_SRC_PEN_TIP | WAKEUP_SRC_CHARGER | WAKEUP_SRC_BUTTON;

    /* 初始化功耗统计 */
    s_active_time_ms = 0;
    s_sleep_time_ms = 0;
}

/* ===== 电源域控制 ===== */

/**

```



```

* 使能指定电源域
* @param domain_mask 电源域掩码
*/
void power_domain_enable(uint8_t domain_mask) {
    if (domain_mask & POWER_DOMAIN_CAMERA) {
        hal_gpio_write(GPIO_CAMERA_POWER, 1);
    }
    if (domain_mask & POWER_DOMAIN_FLASH) {
        hal_gpio_write(GPIO_FLASH_POWER, 1);
    }
    if (domain_mask & POWER_DOMAIN_SENSOR) {
        hal_gpio_write(GPIO_SENSOR_POWER, 1);
    }
    if (domain_mask & POWER_DOMAIN_LED) {
        hal_gpio_write(GPIO_LED_POWER, 1);
    }

    s_active_domains |= domain_mask;
}

/**
* 禁用指定电源域
* @param domain_mask 电源域掩码
*/
void power_domain_disable(uint8_t domain_mask) {
    if (domain_mask & POWER_DOMAIN_CAMERA) {
        hal_gpio_write(GPIO_CAMERA_POWER, 0);
    }
    if (domain_mask & POWER_DOMAIN_FLASH) {
        hal_gpio_write(GPIO_FLASH_POWER, 0);
    }
    if (domain_mask & POWER_DOMAIN_SENSOR) {
        hal_gpio_write(GPIO_SENSOR_POWER, 0);
    }
    if (domain_mask & POWER_DOMAIN_LED) {
        hal_gpio_write(GPIO_LED_POWER, 0);
    }

    s_active_domains &= ~domain_mask;
}

/* ===== 低功耗状态转换 ===== */

/**
* 进入轻度睡眠模式
* - 降低CPU频率到16MHz
* - 关闭摄像头和传感器电源域
* - 保持BLE连接和Flash电源
* - 可由笔尖触摸或BLE事件唤醒
*/
void power_enter_light_sleep(void) {
    /* 关闭不必要的电源域 */
    power_domain_disable(POWER_DOMAIN_CAMERA | POWER_DOMAIN_SENSOR | POWER_DOMAIN_LED);

    /* 降低CPU频率 */
    SystemClock_SetFrequency(LIGHT_SLEEP_FREQ_MHZ);
}

```

```

    /* 配置唤醒源 */
    hal_gpio_set_wakeup(GPIO_PEN_TIP_PIN, GPIO_WAKEUP_RISING);
    hal_gpio_set_wakeup(GPIO_BUTTON_PIN, GPIO_WAKEUP_FALLING);

    /* 进入CPU SLEEP模式（WFI等待中断） */
    __WFI();

    /* 唤醒后恢复 */
    SystemClock_SetFrequency(ACTIVE_FREQ_MHZ);
    power_domain_enable(POWER_DOMAIN_SENSOR | POWER_DOMAIN_LED);
}

/**
 * 进入深度睡眠模式
 * - 关闭所有外设电源域
 * - 断开BLE连接
 * - MCU进入STOP/STANDBY模式
 * - 仅保留RTC和GPIO唤醒
 * - 唤醒后相当于系统复位重启
 */
void power_enter_deep_sleep(void) {
    /* 保存关键数据到Flash */
    save_power_state();

    /* 关闭所有电源域 */
    power_domain_disable(POWER_DOMAIN_ALL);

    /* 配置RTC唤醒（定时检查电量） */
    hal_rtc_set_alarm(RTC_WAKEUP_INTERVAL_S);

    /* 配置GPIO唤醒源 */
    hal_gpio_set_wakeup(GPIO_PEN_TIP_PIN, GPIO_WAKEUP_RISING);
    hal_gpio_set_wakeup(GPIO_USB_DETECT_PIN, GPIO_WAKEUP_RISING);
    hal_gpio_set_wakeup(GPIO_BUTTON_PIN, GPIO_WAKEUP_FALLING);

    /* 进入STANDBY模式（最低功耗，唤醒后从头执行） */
    hal_enter_standby_mode();
}

/* ===== 功耗状态保存/恢复 ===== */

/* Flash中保存电源状态的地址 */
#define POWER_STATE_FLASH_ADDR 0x0000F000

/* 电源状态保存结构 */
typedef struct {
    uint32_t magic; /* 魔数 0xPWR55AA */
    uint32_t total_active_ms; /* 累计活跃时长 */
    uint32_t total_sleep_ms; /* 累计睡眠时长 */
    uint32_t boot_count; /* 启动次数 */
    uint32_t last_shutdown_reason; /* 上次关机原因 */
    uint32_t checksum; /* CRC校验 */
} PowerStateFlash;

/**
 * 保存电源状态到Flash
 * 在进入深度睡眠前调用

```

```

*/
static void save_power_state(void) {
    PowerStateFlash state;
    state.magic = 0x50575200; /* "PWR\0" */
    state.total_active_ms = s_active_time_ms;
    state.total_sleep_ms = s_sleep_time_ms;
    state.boot_count = 0; /* 将在恢复时递增 */
    state.last_shutdown_reason = 0;

    /* 计算校验和 */
    uint32_t sum = 0;
    const uint32_t *data = (const uint32_t *)&state;
    uint8_t i;
    for (i = 0; i < (sizeof(state) / 4) - 1; i++) {
        sum ^= data[i];
    }
    state.checksum = sum;

    /* 写入Flash */
    hal_flash_erase_sector(POWER_STATE_FLASH_ADDR);
    hal_flash_write(POWER_STATE_FLASH_ADDR, (const uint8_t *)&state, sizeof(state));
}

/**
 * 从Flash恢复电源状态
 * 在启动时调用
 */
void power_restore_state(void) {
    PowerStateFlash state;
    hal_flash_read(POWER_STATE_FLASH_ADDR, (uint8_t *)&state, sizeof(state));

    if (state.magic != 0x50575200) {
        /* 无效的保存数据 */
        return;
    }

    /* 验证校验和 */
    uint32_t sum = 0;
    const uint32_t *data = (const uint32_t *)&state;
    uint8_t i;
    for (i = 0; i < (sizeof(state) / 4) - 1; i++) {
        sum ^= data[i];
    }

    if (sum != state.checksum) {
        return; /* 数据损坏 */
    }

    /* 恢复功耗统计 */
    s_active_time_ms = state.total_active_ms;
    s_sleep_time_ms = state.total_sleep_ms;
}

/* ===== 功耗统计接口 ===== */

/**
 * 更新活跃时间统计

```

```

    * @param elapsed_ms 经过的毫秒数
    */
void power_update_active_time(uint32_t elapsed_ms) {
    s_active_time_ms += elapsed_ms;
}

/**
 * 更新睡眠时间统计
 * @param elapsed_ms 经过的毫秒数
 */
void power_update_sleep_time(uint32_t elapsed_ms) {
    s_sleep_time_ms += elapsed_ms;
}

/**
 * 获取累计活跃时长（秒）
 */
uint32_t power_get_active_seconds(void) {
    return s_active_time_ms / 1000;
}

/**
 * 获取电源效率（活跃时间占比百分比）
 */
uint8_t power_get_efficiency(void) {
    uint32_t total = s_active_time_ms + s_sleep_time_ms;
    if (total == 0) return 100;
    return (uint8_t)((uint64_t)s_active_time_ms * 100 / total);
}

/**
 * 获取当前活跃的电源域掩码
 */
uint8_t power_get_active_domains(void) {
    return s_active_domains;
}

```

task/

task/ble_send_task.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * ble_send_task.c - BLE数据发送任务
 *
 * 功能说明：
 * 1. 从坐标队列获取数据并打包为BLE通知帧
 * 2. 7字节紧凑坐标编码格式
 * 3. 发送速率控制（适配BLE连接间隔）
 * 4. 笔落下/抬起事件通知
 * 5. 设备信息特征值更新（电量/固件版本）
 */

```

```

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

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "event_groups.h"

#include "ble_gatt_server.h"

/* ===== BLE帧格式定义 ===== */

/* 帧头标识 */
#define BLE_FRAME_HEADER      0xAA55

/* 帧类型 */
#define FRAME_TYPE_COORDINATE  0x00    /* 坐标数据帧 */
#define FRAME_TYPE_PEN_DOWN    0x01    /* 笔落下事件 */
#define FRAME_TYPE_PEN_UP      0x02    /* 笔抬起事件 */
#define FRAME_TYPE_DEVICE_INFO 0x03    /* 设备信息帧 */
#define FRAME_TYPE_PAGE_CHANGE 0x04    /* 翻页事件 */

/* 最大BLE MTU通知载荷 */
#define BLE_MAX_NOTIFY_SIZE    20

/* 批量发送缓冲区大小（可打包多个坐标点） */
#define BATCH_BUFFER_SIZE      3

/* ===== 外部引用 ===== */

extern QueueHandle_t g_coordinate_queue;
extern EventGroupHandle_t g_ble_event_group;
extern SemaphoreHandle_t g_system_mutex;

/* 坐标数据包结构 */
typedef struct {
    uint32_t raw_x;
    uint32_t raw_y;
    uint16_t pressure;
    uint32_t timestamp_ms;
    uint32_t page_id;
    uint8_t pen_state;
} CoordinatePacket;

/* ===== 静态变量 ===== */

/* 发送缓冲区 */
static uint8_t s_send_buffer[BLE_MAX_NOTIFY_SIZE];

/* BLE连接状态 */
static volatile bool s_ble_connected = false;

/* 当前页面ID（检测翻页） */
static uint32_t s_current_page_id = 0;

/* 发送统计 */

```

```

static uint32_t s_total_sent = 0;
static uint32_t s_send_failures = 0;

/* ===== CRC-16 CCITT计算 ===== */

/**
 * CRC-16 CCITT校验计算
 * 用于BLE传输数据帧的完整性校验
 *
 * @param data    数据缓冲区
 * @param length  数据长度
 * @return CRC-16校验值
 */
static uint16_t crc16_ccitt(const uint8_t *data, uint16_t length) {
    uint16_t crc = 0xFFFF;
    uint16_t i;

    for (i = 0; i < length; i++) {
        crc ^= (uint16_t)data[i] << 8;
        uint8_t j;
        for (j = 0; j < 8; j++) {
            if (crc & 0x8000) {
                crc = (crc << 1) ^ 0x1021;
            } else {
                crc <<= 1;
            }
        }
    }

    return crc;
}

/* ===== 坐标编码 ===== */

/**
 * 将坐标数据编码为7字节紧凑格式
 *
 * 编码格式:
 * 字节0-1: X坐标高16位 (大端序)
 * 字节2-3: Y坐标高16位
 * 字节4:   X低4位(高半字节) | Y低4位(低半字节)
 * 字节5:   压力值高8位
 * 字节6:   压力值低4位(高半字节) | 标志位(低半字节)
 *
 * @param packet  坐标数据包
 * @param output  输出缓冲区 (至少7字节)
 * @param flags   标志位 (低2位: 00=数据, 01=笔落下, 02=笔抬起)
 */
static void encode_coordinate(const CoordinatePacket *packet, uint8_t *output,
                             uint8_t flags) {
    /* X坐标 (20位精度) */
    uint32_t x = packet->raw_x & 0xFFFFF;
    output[0] = (uint8_t)((x >> 12) & 0xFF); /* X高8位 */
    output[1] = (uint8_t)((x >> 4) & 0xFF);   /* X次高8位 */

    /* Y坐标 (20位精度) */
    uint32_t y = packet->raw_y & 0xFFFFF;

```

```

    output[2] = (uint8_t)((y >> 12) & 0xFF);    /* Y高8位 */
    output[3] = (uint8_t)((y >> 4) & 0xFF);    /* Y次高8位 */

    /* X低4位和Y低4位合并到一个字节 */
    output[4] = (uint8_t)(((x & 0x0F) << 4) | (y & 0x0F));

    /* 压力值 (12位精度) */
    uint16_t p = packet->pressure & 0x0FFF;
    output[5] = (uint8_t)((p >> 4) & 0xFF);    /* 压力高8位 */

    /* 压力低4位 | 标志位 */
    output[6] = (uint8_t)(((p & 0x0F) << 4) | (flags & 0x0F));
}

/* ===== BLE通知发送 ===== */

/**
 * 通过BLE GATT通知发送数据帧
 *
 * @param data    帧数据
 * @param length  帧长度
 * @return 0成功, -1未连接, -2发送失败
 */
static int ble_send_notification(const uint8_t *data, uint16_t length) {
    if (!s_ble_connected) {
        return -1;
    }

    /* 调用BLE GATT服务器发送通知 */
    int ret = ble_gatt_notify(BLE_CHAR_STROKE_DATA, data, length);

    if (ret == 0) {
        s_total_sent++;
    } else {
        s_send_failures++;
    }

    return ret;
}

/**
 * 发送笔状态事件 (落下/抬起)
 */
static void send_pen_event(uint8_t event_type) {
    uint8_t frame[7];
    memset(frame, 0, sizeof(frame));

    /* 事件帧只需要标志位, 坐标和压力都为0 */
    frame[6] = event_type & 0x0F;

    ble_send_notification(frame, 7);
}

/**
 * 发送翻页事件
 * 当检测到坐标所在页面发生变化时通知上位机
 */

```

```

static void send_page_change_event(uint32_t new_page_id) {
    uint8_t frame[8];

    frame[0] = FRAME_TYPE_PAGE_CHANGE;
    frame[1] = (uint8_t)((new_page_id >> 24) & 0xFF);
    frame[2] = (uint8_t)((new_page_id >> 16) & 0xFF);
    frame[3] = (uint8_t)((new_page_id >> 8) & 0xFF);
    frame[4] = (uint8_t)(new_page_id & 0xFF);

    /* CRC校验 */
    uint16_t crc = crc16_ccitt(frame, 5);
    frame[5] = (uint8_t)((crc >> 8) & 0xFF);
    frame[6] = (uint8_t)(crc & 0xFF);
    frame[7] = 0;

    ble_send_notification(frame, 8);
}

/**
 * 更新设备信息特征值（电量、固件版本等）
 * 上位机可以随时读取此特征值获取笔的状态
 */
static void update_device_info(uint8_t battery_percent) {
    uint8_t info[4];
    info[0] = battery_percent;          /* 电量百分比 */
    info[1] = 2;                        /* 固件主版本号 */
    info[2] = 1;                        /* 固件次版本号 */
    info[3] = 5;                        /* 固件补丁版本号 → V2.1.5 */

    ble_gatt_update_characteristic(BLE_CHAR_DEVICE_INFO, info, sizeof(info));
}

/* ===== BLE连接事件回调 ===== */

/**
 * BLE连接建立回调（由BLE协议栈调用）
 */
void on_ble_connected(void) {
    s_ble_connected = true;

    BaseType_t higher_priority_woken = pdFALSE;
    xEventGroupSetBitsFromISR(g_ble_event_group, EVT_BLE_CONNECTED,
                              &higher_priority_woken);
    portYIELD_FROM_ISR(higher_priority_woken);
}

/**
 * BLE连接断开回调
 */
void on_ble_disconnected(void) {
    s_ble_connected = false;

    BaseType_t higher_priority_woken = pdFALSE;
    xEventGroupSetBitsFromISR(g_ble_event_group, EVT_BLE_DISCONNECTED,
                              &higher_priority_woken);
    portYIELD_FROM_ISR(higher_priority_woken);
}

```



```

/* ===== BLE发送主任务 ===== */

/**
 * BLE发送任务（FreeRTOS任务函数）
 *
 * 运行流程：
 * 1. 等待BLE连接建立
 * 2. 监听笔状态事件（落下/抬起）并发送事件通知
 * 3. 从坐标队列读取数据，编码为7字节格式发送
 * 4. 翻页检测与通知
 * 5. 定期更新设备信息特征值
 */
void ble_send_task(void *pvParameters) {
    (void)pvParameters;

    CoordinatePacket packet;
    uint32_t info_update_counter = 0;

    /* 注册BLE连接回调 */
    ble_gatt_register_connect_callback(on_ble_connected);
    ble_gatt_register_disconnect_callback(on_ble_disconnected);

    /* 启动BLE广播 */
    ble_gatt_start_advertising();

    while (1) {
        /* 等待BLE连接 */
        if (!s_ble_connected) {
            xEventGroupWaitBits(g_ble_event_group, EVT_BLE_CONNECTED,
                                pdTRUE, pdFALSE, portMAX_DELAY);
        }

        /* 检查笔状态事件 */
        EventBits_t events = xEventGroupGetBits(g_ble_event_group);

        if (events & EVT_PEN_DOWN) {
            xEventGroupClearBits(g_ble_event_group, EVT_PEN_DOWN);
            send_pen_event(FRAME_TYPE_PEN_DOWN);
        }

        if (events & EVT_PEN_UP) {
            xEventGroupClearBits(g_ble_event_group, EVT_PEN_UP);
            send_pen_event(FRAME_TYPE_PEN_UP);
        }

        /* 从坐标队列读取数据（超时10ms，避免永久阻塞） */
        if (xQueueReceive(g_coordinate_queue, &packet, pdMS_TO_TICKS(10)) == pdTRUE) {
            /* 翻页检测 */
            if (packet.page_id != s_current_page_id && s_current_page_id != 0) {
                send_page_change_event(packet.page_id);
            }
            s_current_page_id = packet.page_id;

            /* 编码并发送坐标 */
            uint8_t encoded[7];
            encode_coordinate(&packet, encoded, FRAME_TYPE_COORDINATE);
        }
    }
}

```

```

        ble_send_notification(encoded, 7);
    }

    /* 每500次循环更新一次设备信息（约每5秒） */
    info_update_counter++;
    if (info_update_counter >= 500) {
        info_update_counter = 0;
        /* 读取当前电量 */
        extern uint8_t power_get_battery_percent(void);
        uint8_t battery = power_get_battery_percent();
        update_device_info(battery);
    }
}
}

```

task/coordinate_task.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * coordinate_task.c - 坐标计算任务
 *
 * 功能说明：
 * 1. 从图像帧中解码Anoto点阵图案
 * 2. 计算笔尖在纸面的物理坐标
 * 3. 坐标滤波与去抖（卡尔曼滤波）
 * 4. 坐标打包为BLE传输格式
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

#include "dot_decoder.h"

/* ===== 坐标数据包结构 ===== */

typedef struct {
    uint32_t raw_x;           /* 原始X坐标（20位精度） */
    uint32_t raw_y;           /* 原始Y坐标 */
    uint16_t pressure;         /* 压力值（12位） */
    uint32_t timestamp_ms;    /* 时间戳 */
    uint32_t page_id;         /* 页面ID */
    uint8_t pen_state;         /* 笔状态：0=书写中，1=笔落下，2=笔抬起 */
} CoordinatePacket;

/* ===== 卡尔曼滤波器 ===== */

typedef struct {
    float x_estimate;          /* X状态估计值 */

```

```

    float y_estimate;          /* Y状态估计值 */
    float x_error;             /* X估计误差协方差 */
    float y_error;             /* Y估计误差协方差 */
    float process_noise;       /* 过程噪声 Q */
    float measurement_noise;   /* 测量噪声 R */
    bool initialized;          /* 是否已初始化 */
} KalmanFilter2D;

/* ===== 外部引用 ===== */

extern QueueHandle_t g_image_data_queue;
extern QueueHandle_t g_coordinate_queue;

/* ===== 图像帧元数据结构（与image_capture_task一致） ===== */

typedef struct {
    uint8_t *pixel_buffer;
    uint32_t frame_id;
    uint32_t timestamp_ms;
    uint8_t quality_score;
    uint8_t exposure_value;
    uint16_t pressure_raw;
} ImageFrameMetadata;

/* ===== 静态变量 ===== */

/* 卡尔曼滤波器实例 */
static KalmanFilter2D s_kalman;

/* 上一次有效坐标（用于抖动检测） */
static float s_last_valid_x = 0;
static float s_last_valid_y = 0;

/* 点阵码解码工作缓冲区 */
static uint8_t s_decode_buffer[128];

/* 统计信息 */
static uint32_t s_total_decoded = 0;
static uint32_t s_decode_failures = 0;

/* ===== 卡尔曼滤波实现 ===== */

/**
 * 初始化卡尔曼滤波器
 * @param kf      滤波器实例
 * @param q      过程噪声（越大跟踪越快，噪声越多）
 * @param r      测量噪声（越大滤波越强，延迟越大）
 */
static void kalman_init(KalmanFilter2D *kf, float q, float r) {
    kf->x_estimate = 0;
    kf->y_estimate = 0;
    kf->x_error = 1.0f;
    kf->y_error = 1.0f;
    kf->process_noise = q;
    kf->measurement_noise = r;
    kf->initialized = false;
}

```

```

/**
 * 卡尔曼滤波更新
 * @param kf          滤波器实例
 * @param measured_x  测量X值
 * @param measured_y  测量Y值
 * @param out_x       滤波后X输出
 * @param out_y       滤波后Y输出
 */
static void kalman_update(KalmanFilter2D *kf, float measured_x, float measured_y,
                          float *out_x, float *out_y) {
    if (!kf->initialized) {
        /* 第一次测量, 直接使用测量值 */
        kf->x_estimate = measured_x;
        kf->y_estimate = measured_y;
        kf->initialized = true;
        *out_x = measured_x;
        *out_y = measured_y;
        return;
    }

    /* 预测步骤: 状态不变模型 (笔的位置预测 = 上一次估计) */
    float x_pred = kf->x_estimate;
    float y_pred = kf->y_estimate;
    float x_err_pred = kf->x_error + kf->process_noise;
    float y_err_pred = kf->y_error + kf->process_noise;

    /* 更新步骤: 计算卡尔曼增益 */
    float kx = x_err_pred / (x_err_pred + kf->measurement_noise);
    float ky = y_err_pred / (y_err_pred + kf->measurement_noise);

    /* 融合预测与测量 */
    kf->x_estimate = x_pred + kx * (measured_x - x_pred);
    kf->y_estimate = y_pred + ky * (measured_y - y_pred);

    /* 更新误差协方差 */
    kf->x_error = (1.0f - kx) * x_err_pred;
    kf->y_error = (1.0f - ky) * y_err_pred;

    *out_x = kf->x_estimate;
    *out_y = kf->y_estimate;
}

/**
 * 重置卡尔曼滤波器 (新笔画开始时调用)
 */
static void kalman_reset(KalmanFilter2D *kf) {
    kf->initialized = false;
    kf->x_error = 1.0f;
    kf->y_error = 1.0f;
}

/* ===== 抖动检测 ===== */

/**
 * 检测坐标是否为抖动 (笔静止时传感器的微小抖动)
 * 如果两次坐标之间的距离小于阈值, 视为抖动并丢弃
 */

```

```

*
* @param x          当前X坐标
* @param y          当前Y坐标
* @param threshold 抖动阈值（坐标单位）
* @return true表示是抖动，应丢弃
*/
static bool is_jitter(float x, float y, float threshold) {
    float dx = x - s_last_valid_x;
    float dy = y - s_last_valid_y;
    float distance_sq = dx * dx + dy * dy;

    return (distance_sq < threshold * threshold);
}

/* ===== 点阵码图像解码 ===== */

/**
* 从32x32灰度图像中解码Anoto点阵图案
*
* 解码步骤：
* 1. 二值化：将灰度图转为黑白图
* 2. 点检测：定位图案中的各个墨点位置
* 3. 网格对齐：将检测到的点对齐到规则网格
* 4. 编码读取：根据点相对于网格中心的偏移方向读取编码值
* 5. 坐标计算：将编码序列映射为全局坐标
*
* @param pixels      32x32灰度图像数据
* @param quality      图像质量评分
* @param out_x        解码输出X坐标
* @param out_y        解码输出Y坐标
* @param out_page_id 解码输出页面ID
* @return 0成功，-1解码失败
*/
static int decode_dot_pattern(const uint8_t *pixels, uint8_t quality,
                             uint32_t *out_x, uint32_t *out_y,
                             uint32_t *out_page_id) {

    /* 步骤1：自适应二值化 */
    uint8_t threshold = 128;

    /* 根据图像亮度动态调整阈值（Otsu方法简化版） */
    uint32_t histogram[256] = {0};
    uint16_t i;
    for (i = 0; i < SENSOR_PIXELS; i++) {
        histogram[pixels[i]]++;
    }

    /* 寻找双峰之间的谷值作为阈值 */
    uint32_t total = SENSOR_PIXELS;
    uint32_t sum = 0;
    for (i = 0; i < 256; i++) {
        sum += i * histogram[i];
    }

    uint32_t sum_bg = 0;
    uint32_t weight_bg = 0;
    float max_variance = 0;

```

```

for (i = 0; i < 256; i++) {
    weight_bg += histogram[i];
    if (weight_bg == 0) continue;

    uint32_t weight_fg = total - weight_bg;
    if (weight_fg == 0) break;

    sum_bg += i * histogram[i];
    float mean_bg = (float)sum_bg / weight_bg;
    float mean_fg = (float)(sum - sum_bg) / weight_fg;

    float diff = mean_bg - mean_fg;
    float variance = (float)weight_bg * weight_fg * diff * diff;

    if (variance > max_variance) {
        max_variance = variance;
        threshold = (uint8_t)i;
    }
}

/* 步骤2: 二值化并检测墨点中心 */
uint8_t dot_count = 0;
int16_t dot_x[64], dot_y[64]; /* 最多检测64个点 */

for (int row = 1; row < SENSOR_HEIGHT - 1; row++) {
    for (int col = 1; col < SENSOR_WIDTH - 1; col++) {
        uint8_t center = pixels[row * SENSOR_WIDTH + col];
        if (center < threshold) {
            /* 暗像素, 检查是否为局部极小值 (简单的点中心检测) */
            uint8_t up = pixels[(row - 1) * SENSOR_WIDTH + col];
            uint8_t down = pixels[(row + 1) * SENSOR_WIDTH + col];
            uint8_t left = pixels[row * SENSOR_WIDTH + (col - 1)];
            uint8_t right = pixels[row * SENSOR_WIDTH + (col + 1)];

            if (center <= up && center <= down &&
                center <= left && center <= right) {
                if (dot_count < 64) {
                    dot_x[dot_count] = col;
                    dot_y[dot_count] = row;
                    dot_count++;
                }
            }
        }
    }
}

/* 至少需要检测到4个点才能解码 */
if (dot_count < 4) {
    return -1;
}

/* 步骤3-5: 调用点阵码解码器 (核心算法在dot_decoder模块中) */
DotDecodeResult result;
int ret = dot_decoder_process(dot_x, dot_y, dot_count, &result);

if (ret == 0) {
    *out_x = result.coordinate_x;

```

```

        *out_y = result.coordinate_y;
        *out_page_id = result.page_id;
        return 0;
    }

    return -1;
}

/* ===== 压力值处理 ===== */

/**
 * 处理原始ADC压力值
 * 12位ADC → 归一化并应用非线性映射
 *
 * @param raw_adc 原始ADC值 (0-4095)
 * @return 处理后的压力值 (0-4095, 非线性映射后)
 */
static uint16_t process_pressure(uint16_t raw_adc) {
    /* 去除静态偏移 (笔尖自重产生的基础压力) */
    const uint16_t offset = 200;
    if (raw_adc < offset) {
        return 0;
    }
    uint16_t adjusted = raw_adc - offset;

    /* 非线性映射 (平方根曲线, 使轻触更灵敏) */
    float normalized = (float)adjusted / (4095.0f - offset);
    float mapped = sqrtf(normalized);

    return (uint16_t)(mapped * 4095);
}

/* ===== 坐标计算主任务 ===== */

/**
 * 坐标计算任务 (FreeRTOS任务函数)
 *
 * 运行流程:
 * 1. 从图像数据队列接收帧元数据
 * 2. 解码点阵图案获得原始坐标
 * 3. 卡尔曼滤波去噪
 * 4. 抖动检测
 * 5. 坐标打包并放入BLE发送队列
 */
void coordinate_task(void *pvParameters) {
    (void)pvParameters;

    ImageFrameMetadata frame;
    CoordinatePacket packet;

    /* 初始化卡尔曼滤波器 */
    /* Q=0.1 跟踪速度适中, R=0.5 中等滤波强度 */
    kalman_init(&s_kalman, 0.1f, 0.5f);

    /* 抖动检测阈值 (坐标单位, 约0.1mm) */
    const float jitter_threshold = 3.0f;

```

```

while (1) {
    /* 阻塞等待图像帧数据 */
    if (xQueueReceive(g_image_data_queue, &frame, portMAX_DELAY) != pdTRUE) {
        continue;
    }

    /* 解码点阵图案 */
    uint32_t raw_x, raw_y, page_id;
    int decode_ret = decode_dot_pattern(frame.pixel_buffer, frame.quality_score,
                                        &raw_x, &raw_y, &page_id);

    if (decode_ret != 0) {
        s_decode_failures++;
        continue;
    }
    s_total_decoded++;

    /* 卡尔曼滤波 */
    float filtered_x, filtered_y;
    kalman_update(&s_kalman, (float)raw_x, (float)raw_y,
                 &filtered_x, &filtered_y);

    /* 抖动检测 */
    if (is_jitter(filtered_x, filtered_y, jitter_threshold)) {
        continue; /* 丢弃抖动数据 */
    }

    /* 更新最后有效坐标 */
    s_last_valid_x = filtered_x;
    s_last_valid_y = filtered_y;

    /* 处理压力值 */
    uint16_t pressure = process_pressure(frame.pressure_raw);

    /* 构建坐标数据包 */
    packet.raw_x = (uint32_t)filtered_x;
    packet.raw_y = (uint32_t)filtered_y;
    packet.pressure = pressure;
    packet.timestamp_ms = frame.timestamp_ms;
    packet.page_id = page_id;
    packet.pen_state = 0; /* 书写中 */

    /* 放入BLE发送队列（非阻塞，满则丢弃最老的） */
    if (xQueueSend(g_coordinate_queue, &packet, 0) != pdTRUE) {
        /* 队列满，读出一个旧数据再写入 */
        CoordinatePacket dummy;
        xQueueReceive(g_coordinate_queue, &dummy, 0);
        xQueueSend(g_coordinate_queue, &packet, 0);
    }
}
}

```

task/image_capture_task.c


```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * image_capture_task.c - 图像采集任务
 *
 * 功能说明:
 * 1. 以100Hz频率驱动CMOS图像传感器采集点阵图案
 * 2. DMA方式高速传输图像数据
 * 3. 笔尖接触检测（上升沿/下降沿中断）
 * 4. 图像帧质量快速评估（丢弃模糊帧）
 * 5. 采集参数自适应调节（曝光、增益）
 */

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

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "event_groups.h"

#include "camera_driver.h"
#include "hal_spi.h"
#include "hal_dma.h"
#include "hal_gpio.h"

/* ===== 常量定义 ===== */

/* 采集频率 (Hz) */
#define CAPTURE_FREQUENCY_HZ          100

/* 采集周期 (毫秒) */
#define CAPTURE_PERIOD_MS              (1000 / CAPTURE_FREQUENCY_HZ)

/* 图像传感器分辨率 */
#define SENSOR_WIDTH                   32
#define SENSOR_HEIGHT                  32
#define SENSOR_PIXELS                  (SENSOR_WIDTH * SENSOR_HEIGHT)

/* 单帧图像字节数 (8位灰度) */
#define FRAME_SIZE_BYTES               SENSOR_PIXELS

/* DMA传输缓冲区 (双缓冲) */
#define DMA_BUFFER_COUNT               2

/* 图像质量阈值 (低于此值判定为模糊/无效帧) */
#define QUALITY_THRESHOLD              30

/* 最大连续无效帧数 (超过则认为笔离开纸面) */
#define MAX_INVALID_FRAMES             5

/* 自动曝光调节步长 */
#define AUTO_EXPOSURE_STEP             4

/* ===== 数据结构 ===== */

```

```

/* 图像帧元数据（放入队列传递给坐标计算任务） */
typedef struct {
    uint8_t *pixel_buffer;          /* 指向DMA缓冲区的像素数据 */
    uint32_t frame_id;              /* 帧序号 */
    uint32_t timestamp_ms;          /* 采集时间戳 */
    uint8_t quality_score;          /* 图像质量评分（0-255） */
    uint8_t exposure_value;         /* 当前曝光值 */
    uint16_t pressure_raw;          /* 同步采集的压力原始ADC值 */
} ImageFrameMetadata;

/* 传感器配置参数 */
typedef struct {
    uint8_t exposure;               /* 曝光时间（寄存器值） */
    uint8_t gain;                   /* 模拟增益 */
    uint8_t threshold;              /* 二值化阈值 */
    bool auto_exposure_enabled;     /* 是否启用自动曝光 */
} SensorConfig;

/* ===== 外部引用 ===== */

extern QueueHandle_t g_image_data_queue;
extern EventGroupHandle_t g_ble_event_group;

/* ===== 静态变量 ===== */

/* DMA双缓冲区 */
static uint8_t s_dma_buffer[DMA_BUFFER_COUNT][FRAME_SIZE_BYTES]
    __attribute__((aligned(4)));

/* 当前活跃的DMA缓冲区索引 */
static volatile uint8_t s_active_buffer = 0;

/* 帧计数器 */
static uint32_t s_frame_counter = 0;

/* 连续无效帧计数 */
static uint8_t s_invalid_frame_count = 0;

/* 传感器配置 */
static SensorConfig s_sensor_config;

/* 笔尖状态 */
static volatile bool s_pen_touching = false;

/* ===== 笔尖接触检测 ===== */

/**
 * 笔尖接触检测GPIO中断回调
 * 通过检测微动开关或红外反射判断笔尖是否接触纸面
 */
void pen_tip_irq_handler(void) {
    bool state = hal_gpio_read(GPIO_PEN_TIP_PIN);

    BaseType_t higher_priority_woken = pdFALSE;

    if (state && !s_pen_touching) {
        /* 笔落下（接触纸面） */
    }
}

```

```

        s_pen_touching = true;
        xEventGroupSetBitsFromISR(g_ble_event_group, EVT_PEN_DOWN,
                                   &higher_priority_woken);
    } else if (!state && s_pen_touching) {
        /* 笔抬起（离开纸面） */
        s_pen_touching = false;
        xEventGroupSetBitsFromISR(g_ble_event_group, EVT_PEN_UP,
                                   &higher_priority_woken);
    }

    portYIELD_FROM_ISR(higher_priority_woken);
}

/* ===== DMA传输完成回调 ===== */

/**
 * SPI DMA传输完成中断回调
 * 图像数据已从传感器通过DMA传输到内存缓冲区
 */
void spi_dma_complete_handler(void) {
    /* 切换到另一个DMA缓冲区（乒乓缓冲） */
    s_active_buffer = (s_active_buffer + 1) % DMA_BUFFER_COUNT;
}

/* ===== 图像质量评估 ===== */

/**
 * 快速评估图像帧质量
 * 通过计算图像的对比值（标准差近似）来判断是否有效
 * 有效的点阵图案应该有清晰的明暗对比
 *
 * @param pixels    像素数据
 * @param length    数据长度
 * @return 质量评分（0-255，越高越好）
 */
static uint8_t evaluate_frame_quality(const uint8_t *pixels, uint16_t length) {
    uint32_t sum = 0;
    uint32_t sum_sq = 0;
    uint16_t i;

    /* 采样统计（每4个像素取1个，减少计算量） */
    uint16_t sample_count = 0;
    for (i = 0; i < length; i += 4) {
        uint32_t val = pixels[i];
        sum += val;
        sum_sq += val * val;
        sample_count++;
    }

    if (sample_count == 0) return 0;

    /* 计算方差的近似值（反映对比度） */
    uint32_t mean = sum / sample_count;
    uint32_t variance = (sum_sq / sample_count) - (mean * mean);

    /* 方差映射到0-255评分 */
    if (variance > 2000) return 255;

```

```

        return (uint8_t)(variance * 255 / 2000);
    }

    /* ===== 自动曝光调节 ===== */

    /**
     * 根据图像亮度自动调节传感器曝光参数
     * 目标：使图像平均亮度保持在中间范围（100-150）
     */
    static void auto_exposure_adjust(const uint8_t *pixels, uint16_t length) {
        if (!s_sensor_config.auto_exposure_enabled) {
            return;
        }

        /* 计算平均亮度 */
        uint32_t sum = 0;
        uint16_t i;
        for (i = 0; i < length; i += 8) {
            sum += pixels[i];
        }
        uint8_t avg_brightness = (uint8_t)(sum / (length / 8));

        /* 根据亮度偏差调节曝光值 */
        if (avg_brightness < 80 && s_sensor_config.exposure < 250) {
            /* 过暗，增加曝光 */
            s_sensor_config.exposure += AUTO_EXPOSURE_STEP;
            camera_set_exposure(s_sensor_config.exposure);
        } else if (avg_brightness > 170 && s_sensor_config.exposure > AUTO_EXPOSURE_STEP) {
            /* 过亮，减少曝光 */
            s_sensor_config.exposure -= AUTO_EXPOSURE_STEP;
            camera_set_exposure(s_sensor_config.exposure);
        }
    }

    /* ===== 传感器初始化 ===== */

    /**
     * 配置CMOS图像传感器初始参数
     */
    static void sensor_setup(void) {
        /* 设置默认曝光和增益 */
        s_sensor_config.exposure = 128;
        s_sensor_config.gain = 64;
        s_sensor_config.threshold = 128;
        s_sensor_config.auto_exposure_enabled = true;

        /* 写入传感器寄存器 */
        camera_set_exposure(s_sensor_config.exposure);
        camera_set_gain(s_sensor_config.gain);

        /* 配置为连续帧模式 */
        camera_set_mode(CAMERA_MODE_CONTINUOUS);

        /* 配置SPI DMA接收 */
        hal_dma_config(DMA_CHANNEL_SPI_RX,
                      (uint32_t)camera_get_data_register(),
                      (uint32_t)s_dma_buffer[0],

```

```

        FRAME_SIZE_BYTES);
    }

/* ===== 图像采集主任务 ===== */

/**
 * 图像采集任务 (FreeRTOS任务函数)
 *
 * 运行流程:
 * 1. 等待笔尖接触纸面
 * 2. 以100Hz频率触发CMOS传感器拍摄
 * 3. DMA传输图像数据到双缓冲区
 * 4. 评估图像质量, 丢弃无效帧
 * 5. 将有效帧元数据放入队列供坐标计算任务处理
 * 6. 笔抬起后暂停采集, 进入低功耗等待
 */
void image_capture_task(void *pvParameters) {
    (void)pvParameters;

    TickType_t last_wake_time;

    /* 初始化传感器参数 */
    sensor_setup();

    /* 注册笔尖GPIO中断 */
    hal_gpio_set_irq(GPIO_PEN_TIP_PIN, GPIO_IRQ_BOTH_EDGE, pen_tip_irq_handler);

    while (1) {
        /* 等待笔落下事件 (低功耗阻塞) */
        xEventGroupWaitBits(g_ble_event_group, EVT_PEN_DOWN,
                           pdTRUE, pdFALSE, portMAX_DELAY);

        /* 笔已接触纸面, 启动高速采集 */
        camera_power_on();

        /* 重置帧计数和无效帧计数 */
        s_frame_counter = 0;
        s_invalid_frame_count = 0;

        /* 记录采集起始时间 */
        last_wake_time = xTaskGetTickCount();

        /* 采集循环: 持续采集直到笔抬起 */
        while (s_pen_touching) {
            /* 触发传感器拍摄 */
            camera_trigger_capture();

            /* 启动DMA传输 (异步, CPU可做其他事) */
            uint8_t current_buffer = s_active_buffer;
            hal_dma_start(DMA_CHANNEL_SPI_RX,
                          (uint32_t)s_dma_buffer[current_buffer],
                          FRAME_SIZE_BYTES);

            /* 等待DMA完成 (通常< 1ms) */
            hal_dma_wait_complete(DMA_CHANNEL_SPI_RX, 5);

            /* 同步读取压力传感器ADC值 */

```

```

uint16_t pressure_raw = pressure_sensor_read_raw();

/* 评估图像质量 */
uint8_t quality = evaluate_frame_quality(
    s_dma_buffer[current_buffer], FRAME_SIZE_BYTES);

if (quality >= QUALITY_THRESHOLD) {
    /* 有效帧, 放入队列 */
    ImageFrameMetadata metadata;
    metadata.pixel_buffer = s_dma_buffer[current_buffer];
    metadata.frame_id = s_frame_counter;
    metadata.timestamp_ms = xTaskGetTickCount() * portTICK_PERIOD_MS;
    metadata.quality_score = quality;
    metadata.exposure_value = s_sensor_config.exposure;
    metadata.pressure_raw = pressure_raw;

    /* 非阻塞方式入队 (如果队列满则丢弃) */
    xQueueSend(g_image_data_queue, &metadata, 0);

    s_invalid_frame_count = 0;
} else {
    s_invalid_frame_count++;

    /* 连续多个无效帧, 可能笔已离开但中断未触发 */
    if (s_invalid_frame_count >= MAX_INVALID_FRAMES) {
        s_pen_touching = false;
        break;
    }
}

/* 每16帧调整一次曝光 (避免频繁调节) */
if ((s_frame_counter & 0x0F) == 0) {
    auto_exposure_adjust(s_dma_buffer[current_buffer], FRAME_SIZE_BYTES);
}

s_frame_counter++;

/* 精确定时: 等待到下一个采集时间点 */
vTaskDelayUntil(&last_wake_time, pdMS_TO_TICKS(CAPTURE_PERIOD_MS));
}

/* 笔抬起, 关闭传感器降低功耗 */
camera_power_off();
}
}

```

task/power_monitor_task.c

```

/*
 * 自然写智能点阵笔嵌入式固件软件 V1.0
 * power_monitor_task.c - 电源监测与低功耗管理任务
 *
 * 功能说明:
 * 1. 电池电压ADC采样与电量百分比估算

```

```

* 2. 充电检测与充电状态管理
* 3. 低电量告警与自动关机保护
* 4. 低功耗状态机 (Active → Light Sleep → Deep Sleep)
* 5. USB充电IC状态监测
*/

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

#include "FreeRTOS.h"
#include "task.h"
#include "event_groups.h"
#include "semphr.h"

#include "hal_adc.h"
#include "hal_gpio.h"
#include "power_manager.h"
#include "led_driver.h"

/* ===== 电池参数定义 ===== */

/* 电池满充电压 (mV): 锂聚合物3.7V标称 */
#define BATTERY_FULL_MV          4200

/* 电池截止电压 (mV): 低于此值必须关机保护 */
#define BATTERY_CUTOFF_MV        3300

/* 低电量告警阈值 (百分比) */
#define LOW_BATTERY_THRESHOLD    10

/* 极低电量关机阈值 */
#define CRITICAL_BATTERY_THRESHOLD  3

/* ADC参考电压 (mV) */
#define ADC_VREF_MV              3300

/* ADC分辨率 (12位) */
#define ADC_MAX_VALUE            4095

/* 电池电压分压比 (电阻分压器: R1=100K, R2=100K → 2:1) */
#define BATTERY_DIVIDER_RATIO    2

/* 电压采样滤波窗口大小 */
#define VOLTAGE_FILTER_WINDOW    8

/* 电源监测周期 (毫秒) */
#define POWER_MONITOR_PERIOD_MS  5000

/* 自动休眠超时 (毫秒): 笔静止超过此时间自动进入深度睡眠 */
#define AUTO_SLEEP_TIMEOUT_MS    300000 /* 5分钟 */

/* ===== 电源状态枚举 ===== */

typedef enum {
    POWER_STATE_ACTIVE,          /* 活跃状态 (正常工作) */
    POWER_STATE_LIGHT_SLEEP,     /* 轻度睡眠 (BLE保持连接) */
    POWER_STATE_DEEP_SLEEP,      /* 深度睡眠 (仅保留RTC唤醒) */

```

```

    POWER_STATE_CHARGING,          /* 充电中 */
    POWER_STATE_SHUTDOWN           /* 关机保护 */
} PowerState;

/* ===== 外部引用 ===== */

extern EventGroupHandle_t g_ble_event_group;
extern SemaphoreHandle_t g_system_mutex;

/* 系统状态结构体（在main.c中定义） */
typedef struct {
    bool pen_is_down;
    bool ble_connected;
    bool is_charging;
    uint8_t battery_percent;
    uint32_t total_strokes;
    uint32_t uptime_seconds;
    uint8_t error_flags;
} SystemState;

extern SystemState g_system_state;

/* ===== 静态变量 ===== */

/* 当前电源状态 */
static PowerState s_power_state = POWER_STATE_ACTIVE;

/* 电压采样滤波缓冲区 */
static uint16_t s_voltage_buffer[VOLTAGE_FILTER_WINDOW];
static uint8_t s_voltage_buffer_index = 0;
static bool s_voltage_buffer_full = false;

/* 最后一次活动时间（用于自动休眠判断） */
static uint32_t s_last_activity_time = 0;

/* ===== 电压采样与滤波 ===== */

/**
 * 读取电池原始ADC值并转换为电压（mV）
 */
static uint16_t read_battery_voltage_mv(void) {
    /* 读取ADC原始值 */
    uint16_t adc_raw = hal_adc_read(ADC_CHANNEL_BATTERY);

    /* ADC值 → 分压后电压 → 实际电池电压 */
    uint32_t voltage_mv = (uint32_t)adc_raw * ADC_VREF_MV / ADC_MAX_VALUE;
    voltage_mv *= BATTERY_DIVIDER_RATIO;

    return (uint16_t)voltage_mv;
}

/**
 * 移动平均滤波
 * 对连续采样的电压值取平均，消除ADC噪声
 *
 * @param new_sample 新采样的电压值（mV）
 * @return 滤波后的电压值
 */

```



```

*/
static uint16_t voltage_filter(uint16_t new_sample) {
    s_voltage_buffer[s_voltage_buffer_index] = new_sample;
    s_voltage_buffer_index = (s_voltage_buffer_index + 1) % VOLTAGE_FILTER_WINDOW;

    if (s_voltage_buffer_index == 0) {
        s_voltage_buffer_full = true;
    }

    uint8_t count = s_voltage_buffer_full ? VOLTAGE_FILTER_WINDOW :
s_voltage_buffer_index;
    uint32_t sum = 0;
    uint8_t i;
    for (i = 0; i < count; i++) {
        sum += s_voltage_buffer[i];
    }

    return (uint16_t)(sum / count);
}

/* ===== 电量百分比估算 ===== */

/**
 * 根据电池电压估算电量百分比
 * 使用分段线性插值模拟锂电池放电曲线
 *
 * 锂聚合物电池典型放电曲线（近似分段线性）：
 * 4200mV → 100%
 * 4060mV → 90%
 * 3920mV → 80%
 * 3830mV → 70%
 * 3750mV → 60%
 * 3680mV → 50%
 * 3620mV → 40%
 * 3570mV → 30%
 * 3500mV → 20%
 * 3400mV → 10%
 * 3300mV → 0%
 */
static uint8_t estimate_battery_percent(uint16_t voltage_mv) {
    /* 放电曲线查找表（电压mV → 百分比） */
    static const struct {
        uint16_t voltage;
        uint8_t percent;
    } discharge_curve[] = {
        {4200, 100},
        {4060, 90},
        {3920, 80},
        {3830, 70},
        {3750, 60},
        {3680, 50},
        {3620, 40},
        {3570, 30},
        {3500, 20},
        {3400, 10},
        {3300, 0}
    };
};

```

```

const uint8_t table_size = sizeof(discharge_curve) / sizeof(discharge_curve[0]);

/* 边界检查 */
if (voltage_mv >= discharge_curve[0].voltage) {
    return 100;
}
if (voltage_mv <= discharge_curve[table_size - 1].voltage) {
    return 0;
}

/* 分段线性插值 */
uint8_t i;
for (i = 0; i < table_size - 1; i++) {
    if (voltage_mv >= discharge_curve[i + 1].voltage) {
        uint16_t v_high = discharge_curve[i].voltage;
        uint16_t v_low = discharge_curve[i + 1].voltage;
        uint8_t p_high = discharge_curve[i].percent;
        uint8_t p_low = discharge_curve[i + 1].percent;

        /* 线性插值 */
        uint16_t v_range = v_high - v_low;
        uint16_t v_offset = voltage_mv - v_low;

        return p_low + (uint8_t)((uint32_t)v_offset * (p_high - p_low) / v_range);
    }
}

return 0;
}

/* ===== 充电检测 ===== */

/**
 * 检测USB充电状态
 * 通过GPIO读取充电IC的状态引脚
 *
 * @return 0=未充电, 1=充电中, 2=充满
 */
static uint8_t detect_charging_state(void) {
    /* STAT1引脚: 低电平=充电中, 高电平=充满或未充电 */
    bool stat1 = hal_gpio_read(GPIO_CHARGE_STAT1);

    /* STAT2引脚: 低电平=充满 */
    bool stat2 = hal_gpio_read(GPIO_CHARGE_STAT2);

    /* USB电源检测引脚 */
    bool usb_power = hal_gpio_read(GPIO_USB_DETECT);

    if (!usb_power) {
        return 0; /* USB未连接, 未充电 */
    }

    if (!stat1) {
        return 1; /* 充电中 */
    }
}

```

```

        if (!stat2) {
            return 2;    /* 充满 */
        }

        return 0;
    }

/* ===== LED状态指示 ===== */

/**
 * 根据电源状态和电量更新LED指示
 */
static void update_led_indication(uint8_t battery_percent, uint8_t charge_state) {
    if (charge_state == 1) {
        /* 充电中: 绿色呼吸灯 */
        led_set_mode(LED_MODE_BREATH_GREEN);
    } else if (charge_state == 2) {
        /* 充满: 绿色常亮 */
        led_set_mode(LED_MODE_SOLID_GREEN);
    } else if (battery_percent <= LOW_BATTERY_THRESHOLD) {
        /* 低电量: 红色慢闪 */
        led_set_mode(LED_MODE_BLINK_RED);
    } else if (battery_percent <= CRITICAL_BATTERY_THRESHOLD) {
        /* 极低电量: 红色快闪 */
        led_set_mode(LED_MODE_FAST_BLINK_RED);
    } else if (g_system_state.ble_connected) {
        /* 已连接: 蓝色常亮 */
        led_set_mode(LED_MODE_SOLID_BLUE);
    } else {
        /* 未连接: 蓝色慢闪 */
        led_set_mode(LED_MODE_BLINK_BLUE);
    }
}

/* ===== 低功耗管理 ===== */

/**
 * 进入轻度睡眠模式
 * 关闭不必要的外设, 降低CPU频率
 * BLE连接保持, 可被笔尖触摸或BLE命令唤醒
 */
static void enter_light_sleep(void) {
    if (s_power_state == POWER_STATE_LIGHT_SLEEP) {
        return;
    }

    /* 关闭摄像头 */
    camera_power_off();

    /* 关闭SPI (传感器通信) */
    hal_spi_disable(SPI_PORT_1);

    /* 降低CPU频率到16MHz */
    SystemClock_SetLow();

    /* LED关闭 */
    led_set_mode(LED_MODE_OFF);
}

```

```

    s_power_state = POWER_STATE_LIGHT_SLEEP;
}

/**
 * 进入深度睡眠模式
 * 关闭所有外设和BLE，仅保留RTC和GPIO唤醒
 * 适用于长时间不使用的场景
 */
static void enter_deep_sleep(void) {
    /* 断开BLE连接 */
    ble_gatt_disconnect();
    ble_gatt_stop_advertising();

    /* 关闭所有外设 */
    camera_power_off();
    hal_spi_disable(SPI_PORT_1);
    hal_i2c_disable(I2C_PORT_1);
    hal_adc_disable(ADC_CHANNEL_BATTERY);

    /* 保存系统状态到Flash */
    offline_storage_flush();

    /* 配置唤醒源（笔尖GPIO中断唤醒） */
    hal_gpio_set_wakeup(GPIO_PEN_TIP_PIN, GPIO_WAKEUP_RISING);

    /* 进入MCU深度睡眠模式（不应返回） */
    hal_enter_deep_sleep();
}

/**
 * 从轻度睡眠唤醒，恢复正常工作状态
 */
static void wake_from_light_sleep(void) {
    /* 恢复CPU频率 */
    SystemClock_Config();

    /* 重新使能SPI */
    hal_spi_enable(SPI_PORT_1);

    s_power_state = POWER_STATE_ACTIVE;
    s_last_activity_time = xTaskGetTickCount();
}

/* ===== 电源监测主任务 ===== */

/**
 * 电源监测任务（FreeRTOS任务函数）
 *
 * 运行流程：
 * 1. 定期读取电池电压并估算电量
 * 2. 检测充电状态
 * 3. 低电量告警和自动关机保护
 * 4. 更新LED状态指示
 * 5. 自动休眠判断
 */
void power_monitor_task(void *pvParameters) {

```

```

(void)pvParameters;

TickType_t last_wake_time = xTaskGetTickCount();
s_last_activity_time = last_wake_time;

while (1) {
    /* 读取并滤波电池电压 */
    uint16_t raw_mv = read_battery_voltage_mv();
    uint16_t filtered_mv = voltage_filter(raw_mv);

    /* 估算电量百分比 */
    uint8_t battery_percent = estimate_battery_percent(filtered_mv);

    /* 检测充电状态 */
    uint8_t charge_state = detect_charging_state();

    /* 更新全局系统状态 */
    if (xSemaphoreTake(g_system_mutex, pdMS_TO_TICKS(100)) == pdTRUE) {
        g_system_state.battery_percent = battery_percent;
        g_system_state.is_charging = (charge_state == 1);
        xSemaphoreGive(g_system_mutex);
    }

    /* 更新LED指示 */
    update_led_indication(battery_percent, charge_state);

    /* 低电量告警处理 */
    if (battery_percent <= LOW_BATTERY_THRESHOLD && charge_state == 0) {
        /* 通知上位机低电量 */
        xEventGroupSetBits(g_ble_event_group, EVT_LOW_BATTERY);
    }

    /* 极低电量自动关机保护 */
    if (battery_percent <= CRITICAL_BATTERY_THRESHOLD && charge_state == 0) {
        enter_deep_sleep();
    }

    /* 充电状态变化通知 */
    if (charge_state > 0) {
        xEventGroupSetBits(g_ble_event_group, EVT_CHARGING);
        s_power_state = POWER_STATE_CHARGING;
        s_last_activity_time = xTaskGetTickCount();
    }

    /* 自动休眠检查：笔没有书写且BLE空闲超时 */
    if (!g_system_state.pen_is_down && charge_state == 0) {
        uint32_t idle_time = (xTaskGetTickCount() - s_last_activity_time)
            * portTICK_PERIOD_MS;

        if (idle_time > AUTO_SLEEP_TIMEOUT_MS) {
            if (s_power_state == POWER_STATE_ACTIVE) {
                enter_light_sleep();
            } else if (idle_time > AUTO_SLEEP_TIMEOUT_MS * 2) {
                /* 静止超过10分钟，进入深度睡眠 */
                enter_deep_sleep();
            }
        }
    }
}

```

```

    } else {
        /* 有活动, 重置计时器 */
        s_last_activity_time = xTaskGetTickCount();
        if (s_power_state == POWER_STATE_LIGHT_SLEEP) {
            wake_from_light_sleep();
        }
    }

    /* 休眠到下一个监测周期 */
    vTaskDelayUntil(&last_wake_time, pdMS_TO_TICKS(POWER_MONITOR_PERIOD_MS));
}

/* ===== 外部查询接口 ===== */

/** 获取当前电量百分比 (供其他模块调用) */
uint8_t power_get_battery_percent(void) {
    return g_system_state.battery_percent;
}

/** 获取当前电源状态 */
uint8_t power_get_state(void) {
    return (uint8_t)s_power_state;
}

```