

自然写智能点阵笔嵌入式固件软件 V1.0

软件鉴别材料 — 嵌入式软件设计说明书

软件全称：自然写智能点阵笔嵌入式固件软件

软件版本：V1.0

权利人：深圳自然写科技有限公司

文档类型：嵌入式固件软件设计说明书

文档编号：WRITECH-FIRMWARE-DS-001

编制日期：2026年2月

密级：内部资料

目录

- 第一章 软件整体概述
 - 1.1 软件简介与功能综述
 - 1.2 软件用途与适用场景
 - 1.3 运行环境与硬件平台
 - 1.4 开发语言与工具链
 - 1.5 版本说明
- 第二章 系统架构与设计思路
 - 2.1 总体架构设计
 - 2.2 RTOS任务模型
 - 2.3 各层次详细说明
 - 2.4 数据设计
 - 2.5 接口设计 (BLE GATT)
 - 2.6 安全设计
 - 2.7 Flash分区规划
- 第三章 核心模块功能详细说明
 - 3.1 main.c — 主程序与RTOS启动
 - 3.2 driver/camera.c — 点阵摄像头驱动
 - 3.3 driver/pressure.c — 压力传感器驱动
 - 3.4 driver/battery.c — 电池电量监测驱动

- 3.5 codec/dot_decoder.c — 点阵码坐标解码
- 3.6 codec/stroke_encoder.c — 笔迹数据编码打包
- 3.7 task/image_capture_task.c — 图像采集任务
- 3.8 task/coord_calc_task.c — 坐标计算任务
- 3.9 task/ble_send_task.c — BLE数据发送任务
- 3.10 task/power_task.c — 电源管理任务
- 3.11 task/led_task.c — LED状态指示任务
- 3.12 task/ota_task.c — OTA固件升级任务
- 3.13 cache/offline_cache.c — 离线数据缓存
- 3.14 power/power_manager.c — 低功耗状态机
- 第四章 操作流程与使用步骤
- 4.1 点阵笔开机流程
- 4.2 蓝牙配对与连接流程
- 4.3 书写采集与数据传输流程
- 4.4 离线书写与数据同步流程
- 4.5 充电与电量管理
- 4.6 OTA固件升级流程
- 4.7 出厂校准与测试流程
- 4.8 常见问题处理
- 第五章 与源代码的对应关系
- 5.1 模块与源代码文件对应表
- 5.2 核心函数说明
- 5.3 寄存器与GATT特征定义
- 附录A BLE GATT服务定义表
- 附录B 硬件外设寄存器说明
- 附录C 术语表
- 附录D 版本历史

第一章 软件整体概述

1.1 软件简介与功能综述

自然写智能点阵笔嵌入式固件软件（以下简称“笔固件”）是运行于自然写智能点阵笔主控MCU芯片上的嵌入式实时操作系统软件，是整个互动课堂系统最基础的数据采集端软件。笔固件负责控制点阵摄像头连续采集点阵纸面图像，对图像进行实时解码以获取精确的书写坐标，并通过蓝牙BLE协议将坐标流实时发送至网关或终端设备。

笔固件采用RTOS（FreeRTOS / RT-Thread）实时操作系统，以多任务并发的方式同时处理图像采集、坐标解算、蓝牙发送、电源管理、LED指示等功能，各任务按优先级调度，确保100Hz的高频坐标采样率和低延迟蓝牙传输。

主要功能模块综述：

功能模块	说明
点阵摄像头图像采集	控制CMOS摄像头以100fps速率连续采集点阵图像
点阵码坐标解码	对采集图像实时解码，解算出高精度书写坐标（分辨率0.01mm）
压力传感器数据采集	读取笔尖压力传感器ADC值，检测落笔/抬笔事件
BLE数据传输	通过BLE 5.0 GATT Notify方式将坐标流发送至连接的设备
设备配对与连接管理	管理BLE连接配对，支持存储最多4个已配对设备
低功耗电源管理	实现Active/Idle/Sleep/DeepSleep四级电源状态，延长续航
电池电量监测	ADC采样电池电压，计算电量百分比，低电量提示
LED状态指示	通过RGB LED显示连接状态、充电状态、电量告警
离线数据缓存	无连接时在外Flash缓存笔迹数据（容量4MB）
OTA固件升级	支持通过BLE DFU协议接收固件包并安全升级

1.2 软件用途与适用场景

笔固件专为互动课堂智能点阵笔设计，支持以下使用场景：

场景一：课堂实时书写 学生使用点阵笔在配套点阵纸上书写作业、答题或练字。笔固件以100Hz频率采集坐标，通过BLE实时传输至网关或算力盒，实现云端或边缘AI的即时识别与反馈。

场景二：离线书写缓存 当BLE未连接（如课前预习、课后作业）时，笔固件将书写坐标缓存至外部Flash存储（最多约10万个坐标点，约相当于10页A4纸的书写量）。当与设备重新连接后，自动同步缓存数据。

场景三：移动教学（教师手持） 教师手景点阵笔在任意位置书写，笔固件通过BLE将笔迹实时传输至教师手机APP，实现移动式板书和批注。

场景四：字帖练字 配合练字字帖，笔固件高精度采集用户笔顺和坐标，由上层软件进行笔顺分析和书写规范性评测。

适用硬件：

硬件参数	规格
主控MCU	Nordic Semiconductor nRF52840 / STM32WB55
BLE协议版本	Bluetooth Low Energy 5.0
摄像头	定制CMOS点阵摄像头模组（DFOV 20°，100fps）
压力传感器	电阻式力敏传感器，量程0–150g，12位ADC
外部Flash	SPI NOR Flash，4MB（如W25Q32）
电池	锂电池 150mAh，续航约8小时（连接状态）
充电	USB Type-C，500mA充电电流

1.3 运行环境与硬件平台

主控芯片规格（以nRF52840为例）：

项目	规格
处理器架构	ARM Cortex-M4F，64MHz
内部SRAM	256KB
内部Flash	1MB（Bootloader + App A + App B + NVS）
BLE协议栈	Nordic SoftDevice S140（BLE 5.0）
外设接口	SPI×3、I2C×2、ADC 12位×8通道、UART×2、GPIO
电源范围	1.7V ~ 5.5V
低功耗模式	System ON Sleep：1.5μA；System OFF：0.2μA

RTOS运行要求：

组件	版本 / 规格
FreeRTOS	V10.4.3（或 RT-Thread 4.1.0）
Nordic SoftDevice	S140 v7.3.0（BLE协议栈）
最小栈空间	每任务最小512字节栈
总SRAM需求	约180KB（含协议栈、任务栈、数据缓冲）

外部硬件接口：

外设	接口	连接说明
点阵摄像头	SPI（最高8MHz）	图像数据读取（每帧~1KB）
摄像头控制	GPIO（3引脚）	电源使能/帧同步/曝光控制
压力传感器	ADC（12位）	笔尖压力模拟量采样
外部Flash	SPI（最高50MHz）	离线坐标缓存
充电管理IC	I2C	充电状态与电池电压读取
LED	GPIO（RGB三色）	状态指示
振动马达	GPIO	反馈振动（配对成功/低电量提示）

1.4 开发语言与工具链

开发语言：

语言	标准	用途
C	C99 / C11	全部固件源代码
汇编	ARM Thumb-2	中断向量表、启动文件（startup_nrf52840.s）

工具链：

工具	版本	说明
ARM GCC	11.2-2022.02	C/汇编编译器
GNU Make	4.3	构建系统
nRF5 SDK	17.1.0	Nordic嵌入式SDK（BLE、驱动、HAL）
J-Link	V7.80	调试器（SWD接口）
nRF Connect	4.0	BLE调试与协议分析工具
OpenOCD	0.11.0	开源调试接口（备用）
Python	3.9	固件打包脚本、Flash烧写工具

代码规范： – 命名：宏定义全大写下划线（MAX_CONNECTIONS），函数小写下划线（ble_send_data），全局变量 g_ 前缀，静态变量 s_ 前缀 – 中断服务程序（ISR）中禁止调用任何RTOS阻塞API – 所有硬件访问通过驱动层封装，禁止在业务层直接操作寄存器 – 每个源文件不超过500行，超过需拆分

1.5 版本说明

版本	日期	主要变更
V0.3 Alpha	2025年6月	基础BLE连接、坐标采集与发送
V0.7 Beta	2025年9月	离线缓存、点阵码解码算法优化（精度提升）
V0.9 RC	2025年11月	OTA升级、低功耗优化（续航延长30%）
V1.0	2026年2月	正式版：安全加固、压力感应优化、LED动效

第二章 系统架构与设计思路

2.1 总体架构设计

笔固件采用经典的嵌入式RTOS分层架构，自下而上分为五层：硬件抽象层、驱动层、协议栈层、应用任务层和系统管理层。各层之间通过明确定义的API接口通信，保证层间解耦，便于移植和维护。

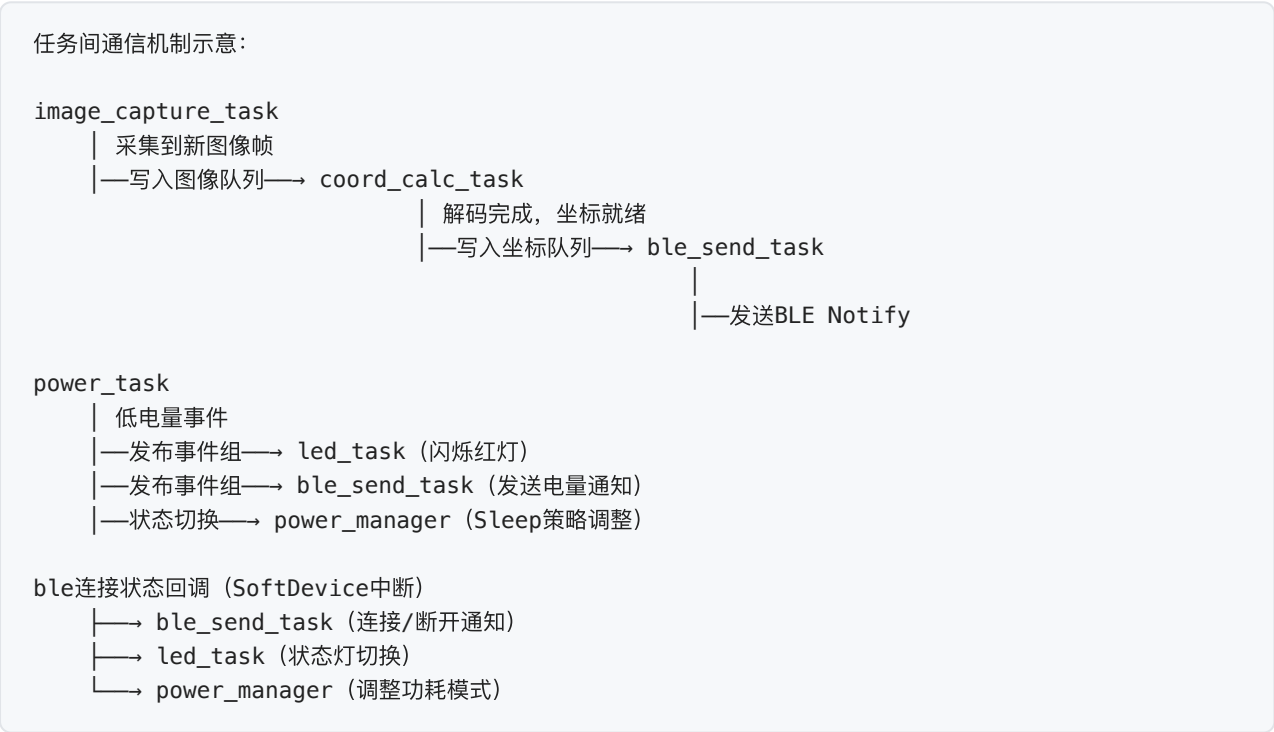
应用任务层 (Application Tasks)				
图像采集任务	坐标计算任务	BLE发送任务	电源监测任务	OTA任务
系统管理层 (System Management Layer)				
配对管理	离线缓存管理	LED控制	日志/错误处理	
BLE协议栈层		图像处理层		
Nordic SoftDevice S140		点阵码解码算法 (dot_decoder.c)		
GATT Server / BLE SMP		笔画编码打包 (stroke_encoder.c)		
硬件驱动层 (Driver Layer)				
摄像头驱动	压力传感器驱动	Flash驱动	充电IC驱动	LED驱动
硬件抽象层 (HAL / nRF SDK)				
GPIO HAL	SPI HAL	ADC HAL	I2C HAL	Timer HAL
硬件层 (MCU + 外设)				
nRF52840 ARM M4F	CMOS摄像头	Flash	传感器	BLE天线

2.2 RTOS任务模型

笔固件运行6个RTOS并发任务，各任务独立调度，通过队列、信号量和事件组进行同步：

任务名称	优先级	栈大小	调度方式	说明
image_capture_task	最高 (5)	1024B	100Hz定时触发	摄像头图像采集
coord_calc_task	高 (4)	2048B	事件触发 (图像就绪)	点阵码解码与坐标计算
ble_send_task	高 (4)	1024B	事件触发 (坐标就绪)	BLE Notify发送
power_task	中 (3)	512B	1Hz定时	电量采样与功耗管理
led_task	低 (2)	512B	事件触发	LED状态控制
ota_task	最低 (1)	4096B	触发式 (BLE DFU命令)	OTA固件升级

任务间通信机制：



2.3 各层次详细说明

硬件抽象层 (HAL)

硬件抽象层基于Nordic nRF5 SDK提供的HAL API，封装了GPIO、SPI、ADC、I2C等基础外设操作。所有驱动层代码通过HAL接口访问硬件，不直接操作寄存器，保证代码可移植性。

硬件驱动层 (Driver Layer)

驱动层为每个外设提供独立的C模块： - driver/camera.c：CMOS摄像头驱动（SPI读取图像帧数据，GPIO控制曝光） - driver/pressure.c：压力传感器ADC采样驱动（12位ADC，过采样16次取均值） - driver/battery.c：电池电量检测（分压电路ADC采样，查表法计算电量百分比）

- driver/flash.c：外部SPI NOR Flash驱动（页读写、扇区擦除、磨损均衡） -
- driver/led.c：RGB LED驱动（PWM控制色彩，支持呼吸灯、闪烁等动效）

BLE协议栈层

采用Nordic SoftDevice S140作为BLE 5.0协议栈，以软件"协议栈"方式与应用代码共存于同一芯片，通过SVC调用（Supervisor Call）接口访问。

GATT Server定义了3个自定义Service： - **Writtech Pen Data Service** (UUID: FFF0)：笔迹坐标数据传输 - **Writtech Device Info Service** (UUID: FFF1)：设备信息读取 - **Writtech DFU Service** (UUID: FFF2)：OTA固件升级

图像处理层

图像处理层实现点阵码图案的识别与坐标解算： - codec/dot_decoder.c：从摄像头原始灰度图像中识别Anoto点阵码图案，解算出全球唯一的纸面坐标（精度0.01mm，分辨率600DPI） - codec/stroke_encoder.c：将坐标数据打包为压缩二进制格式，支持差分编码（降低数据量约60%）

应用任务层

应用任务层运行各业务RTOS任务，处理各功能的具体业务逻辑。每个任务在独立栈空间中运行，通过FreeRTOS队列和事件组进行协作。

2.4 数据设计

Flash分区规划：

nRF52840 内部Flash (1MB = 0x00100000) 分区布局：

地址范围	大小	用途
0x00000000-0x00026FFF	156KB	Nordic SoftDevice S140 (BLE协议栈)
0x00027000-0x0002FFFF	36KB	Bootloader (含OTA引导逻辑)
0x00030000-0x0008FFFF	384KB	Application分区A (当前运行)
0x00090000-0x000EFFFF	384KB	Application分区B (OTA升级目标分区)
0x000F0000-0x000FDFFF	56KB	NVS (非易失性存储：配对信息/设备配置)
0x000FE000-0x000FFFFF	8KB	保留 (厂商信息/校准参数)

外部Flash分区 (4MB SPI NOR Flash)：

外部SPI NOR Flash 分区布局 (4MB = 0x00400000)：

地址范围	大小	用途
-----	-----	-----

0x000000-0x3EFFFF	~4MB	离线坐标缓存 (FIFO环形队列)
0x3F0000-0x3FFFFFF	64KB	保留 (扩展配置/校准数据)

内存 (SRAM 256KB) 使用分配：

区域	大小	说明
SoftDevice保留	约64KB	BLE协议栈内部使用
图像缓冲区 (双缓冲)	2 × 4KB = 8KB	摄像头图像帧数据 (乒乓缓冲)
坐标队列	2KB	待发送坐标数据队列 (FreeRTOS队列)
离线缓存写缓冲	4KB	写入外部Flash的批量缓冲
BLE发送缓冲	1KB	BLE Notify待发送数据包队列
任务栈合计	约9KB	6个任务的栈空间之和
全局变量/堆	约24KB	驱动状态、RTOS对象、临时缓冲

核心数据结构 (C结构体)：

```
/* 坐标数据帧 (coord_data.h) */
/* 每帧7字节, 100Hz采样率下每秒700字节 */
typedef struct {
    uint16_t x;           /* 点阵X坐标 (0~65535, 精度0.01mm) */
    uint16_t y;           /* 点阵Y坐标 (0~65535, 精度0.01mm) */
    uint8_t pressure;     /* 笔尖压力 (0~255, 0=抬笔) */
    uint8_t seq;          /* 序列号 (0~255循环, 用于丢包检测) */
    uint8_t flags;        /* 标志位: bit0=pen_up, bit1=battery_low */
} coord_frame_t;         /* 总大小: 7字节 */

/* BLE数据包 (每包最多包含34个坐标帧, 238字节, 适配BLE MTU=247) */
typedef struct {
    uint8_t packet_type;  /* 0x01=坐标数据, 0x02=电量, 0x03=离线同步 */
    uint8_t frame_count;  /* 本包中坐标帧数量 (1~34) */
    uint16_t base_timestamp; /* 本包第一帧的时间戳低16位 (毫秒) */
    coord_frame_t frames[34]; /* 坐标帧数组 */
} ble_stroke_packet_t;

/* NVS存储的配对信息 (per_bond.h) */
typedef struct {
    uint8_t peer_addr[6]; /* 配对设备蓝牙地址 */
    uint8_t ltk[16];      /* Long-Term Key (加密密钥) */
    uint8_t irk[16];      /* Identity Resolving Key */
    uint8_t peer_name[20]; /* 设备名称 (可选) */
    uint32_t last_connect_ts; /* 最后连接时间戳 (Unix秒) */
} bond_info_t;           /* 最多存储4条配对记录 */

/* 外部Flash离线缓存帧头 (flash_cache.h) */
typedef struct {
    uint32_t magic;        /* 魔数: 0xAA55CC33, 用于帧对齐检测 */
    // ... (other fields) ...
}
```

```
uint32_t timestamp;      /* 书写时间戳 (Unix秒) */
uint16_t frame_count;    /* 本组帧数量 */
uint16_t crc16;          /* 本组数据CRC16校验和 */
} cache_group_header_t;
```

2.5 接口设计 (BLE GATT)

GATT Service和Characteristic定义:

Service 1: Writtech Pen Data Service (UUID: FFF0)

Characteristic	UUID	属性	说明
Stroke Data	FFF1	Notify	实时笔迹坐标流 (每包1~34帧坐标)
Pen Control	FFF2	Write	接收控制指令 (开始/停止采集/请求电量)
Battery Level	FFF3	Read / Notify	电池电量 (0~100%, 低电量时主动Notify)

Service 2: Writtech Device Info Service (UUID: FEE0)

Characteristic	UUID	属性	说明
Device Serial	FEE1	Read	设备序列号 (16字节ASCII)
Firmware Version	FEE2	Read	固件版本号 (如"V1.0.0")
Hardware Version	FEE3	Read	硬件版本号 (如"HW_R1.2")
Calibration Params	FEE4	Read / Write	摄像头校准参数 (出厂写入, 可刷新)

Service 3: Writtech DFU Service (UUID: FEF0)

Characteristic	UUID	属性	说明
DFU Control	FEF1	Write / Indicate	OTA控制 (开始/暂停/取消/确认)
DFU Packet	FEF2	Write Without Response	固件包分块传输 (每块20字节)
DFU Status	FEF3	Indicate	OTA进度上报 (百分比/错误码)

BLE广播数据包格式:

ADV_IND广播包:

- └─ Flags: LE General Discoverable Mode, BR/EDR Not Supported
- └─ Complete Local Name: "Writtech-XXXXXX" (后6位为设备序列号后缀)
- └─ 16-bit Service UUID: FFF0 (Writtech Pen Data Service)
- └─ Manufacturer Specific Data:

```
byte[0-1]: 公司ID (深圳自然写科技, 0x0FF2)
byte[2]:   设备类型 (0x01=点阵笔)
byte[3]:   电量 (0~100)
byte[4]:   固件主版本号
byte[5]:   连接状态 (bit0=是否已连接)
```

2.6 安全设计

BLE连接安全 (LE Secure Connections):

笔固件采用BLE 5.0 LE Secure Connections配对方案: – 使用ECDH (Elliptic Curve Diffie-Hellman) 密钥交换生成配对密钥 – 配对方式: Numeric Comparison (数字比对), 用户在设备端确认6位数字一致 – 配对完成后建立LTK (Long-Term Key), 后续重连使用LTK直接加密, 无需重新配对 – LTK存储在内部Flash NVS区域, 启用读保护防止被读取

固件安全 (Flash读保护):

```
/* 启用APPROTECT (访问端口保护), 防止调试器读取Flash内容 */
/* 在Bootloader中设置: UICR.APPROTECT = 0xFFFFFFFF00 */
#define ENABLE_APPROTECT_ON_PRODUCTION 1

#if ENABLE_APPROTECT_ON_PRODUCTION
static void enable_flash_protection(void) {
    /* 如果APPROTECT未锁定, 执行锁定并重启 */
    if (NRF_UICR->APPROTECT != 0xFFFFFFFF00) {
        nrf_nvmc_uicr_write(&NRF_UICR->APPROTECT, 0xFFFFFFFF00);
        NVIC_SystemReset();
    }
}
#endif
```

OTA升级安全:

固件包安全验证流程 (ota_task.c):

1. 接收完整固件包 (BLE分块传输)
2. CRC32文件完整性校验 (防止传输损坏)
3. RSA-2048签名验证 (使用烧录在NVS中的厂商公钥)
4. 版本号校验 (防止降级攻击, 新版本号必须 > 当前版本)
5. 写入B分区 (App B区)
6. 设置A/B引导标志, 下次重启从B分区启动
7. 重启后验证B分区CRC (Bootloader执行)
8. 验证通过→正式切换; 验证失败→清除B分区标志, 继续从A分区启动

离线缓存数据完整性:

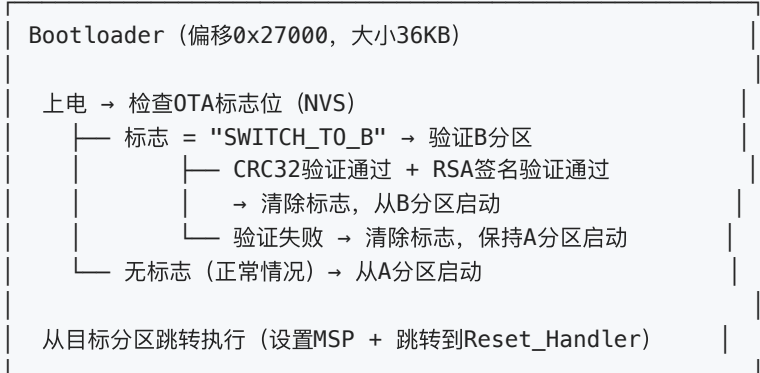
每个离线缓存组写入外部Flash时计算CRC16校验和, 读取时验证。若发现数据损坏 (CRC不匹配), 自动跳过该组, 防止坏数据污染后续数据同步。

看门狗 (Watchdog Timer):

硬件看门狗定时器配置为32秒超时。软件在主任务中每10秒执行一次"喂狗"操作。若固件因死锁或异常未能及时喂狗，看门狗触发硬件复位，恢复正常运行。

2.7 Flash分区规划

内部Flash Bootloader设计:



第三章 核心模块功能详细说明

3.1 main.c — 主程序与RTOS启动

main.c 是固件的启动入口，负责硬件初始化、BLE协议栈配置和RTOS任务创建。

启动流程:

```
int main(void) {
    /* 1. 时钟系统初始化 (HFX0 32MHz晶振, LFX0 32.768kHz晶振) */
    clocks_init();

    /* 2. 日志系统初始化 (RTT日志, 调试期使用) */
    log_init();

    /* 3. GPIO初始化 (摄像头使能脚、LED、振动马达) */
    gpio_init();

    /* 4. SPI总线初始化 (摄像头SPI + Flash SPI) */
    spi_init();

    /* 5. ADC初始化 (压力传感器 + 电池电压) */
    adc_init();

    /* 6. 外部Flash驱动初始化 (挂载离线缓存文件系统) */
    flash_driver_init();
}
```

```

offline_cache_init();

/* 7. BLE协议栈初始化 (SoftDevice使能) */
ble_stack_init();

/* 8. GATT Service初始化 (注册3个自定义Service) */
gatt_services_init();

/* 9. 广播初始化 (配置ADV数据包, 不立即开始广播) */
advertising_init();

/* 10. NVS初始化 (读取配对信息 + 设备配置) */
nvs_init();
peer_manager_init();

/* 11. 电源管理初始化 */
power_manager_init();

/* 12. 看门狗定时器初始化 (32秒超时) */
wdt_init();

/* 13. 创建RTOS任务 */
xTaskCreate(image_capture_task, "IMG", 1024, NULL, 5, NULL);
xTaskCreate(coord_calc_task, "CORD", 2048, NULL, 4, NULL);
xTaskCreate(ble_send_task, "BLE", 1024, NULL, 4, NULL);
xTaskCreate(power_task, "PWR", 512, NULL, 3, NULL);
xTaskCreate(led_task, "LED", 512, NULL, 2, NULL);
xTaskCreate(ota_task, "OTA", 4096, NULL, 1, NULL);

/* 14. 开始BLE广播 */
advertising_start();

/* 15. 启动RTOS调度器 (此后main不再返回) */
vTaskStartScheduler();

/* 不应到达此处 */
for(;;);
}

```

BLE协议栈初始化 (ble_stack_init):

```

/* main.c - BLE协议栈初始化 */
static void ble_stack_init(void) {
    uint32_t err_code;
    uint32_t ram_start = 0;

    /* 使能SoftDevice (指定低频时钟源: 外部32.768kHz晶振) */
    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    /* 配置SoftDevice BLE参数 */
    ble_cfg_t ble_cfg;
    memset(&ble_cfg, 0, sizeof(ble_cfg));

    /* 最大连接数: 1个主设备 + 0个从设备 */

```

```

ble_cfg.conn_cfg.conn_cfg_tag = APP_BLE_CONN_CFG_TAG;
ble_cfg.conn_cfg.params.gap_conn_cfg.conn_count = 1;
ble_cfg.conn_cfg.params.gap_conn_cfg.event_length = 6;
err_code = sd_ble_cfg_set(BLE_CONN_CFG_GAP, &ble_cfg, ram_start);
APP_ERROR_CHECK(err_code);

/* 使能SoftDevice BLE协议栈 */
err_code = nrf_sdh_ble_enable(&ram_start);
APP_ERROR_CHECK(err_code);

/* 注册BLE事件处理回调 */
NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO,
                      ble_evt_handler, NULL);
}

```

3.2 driver/camera.c — 点阵摄像头驱动

点阵摄像头驱动控制定制CMOS摄像头模组，实现100fps高频图像采集。

摄像头初始化序列：

```

/* driver/camera.c */
#define CAMERA_SPI_INSTANCE      0          /* SPI0 */
#define CAMERA_CS_PIN            NRF_GPIO_PIN_MAP(0, 3)
#define CAMERA_VSYNC_PIN        NRF_GPIO_PIN_MAP(0, 4)
#define CAMERA_PWDN_PIN         NRF_GPIO_PIN_MAP(0, 5)
#define CAMERA_IMAGE_SIZE       1024      /* 每帧图像字节数 (32×32像素灰度图) */

/* 摄像头初始化（通过SPI写入寄存器配置序列） */
ret_code_t camera_init(void) {
    ret_code_t err;

    /* 1. 拉低PWDN脚（上电使能摄像头） */
    nrf_gpio_pin_clear(CAMERA_PWDN_PIN);
    nrf_delay_ms(10); /* 等待摄像头稳定 */

    /* 2. 软复位 */
    camera_write_reg(REG_RESET, 0x80);
    nrf_delay_ms(5);

    /* 3. 配置曝光时间（针对点阵纸反射强度优化） */
    camera_write_reg(REG_EXPOSURE_H, 0x00);
    camera_write_reg(REG_EXPOSURE_L, 0x64); /* 曝光时间约100us */

    /* 4. 配置增益（自动增益控制使能） */
    camera_write_reg(REG_GAIN_CTRL, 0x07); /* AGC使能，最大增益×8 */

    /* 5. 配置帧率（100fps: PCLK分频系数） */
    camera_write_reg(REG_CLKDIV, 0x01); /* 不分频，最大帧率 */

    /* 6. 配置输出格式（8位灰度，32×32像素） */
    camera_write_reg(REG_FORMAT, 0x00); /* 灰度模式 */

    /* 7. 验证ID寄存器（防止硬件不兼容） */
}

```

```

uint8_t chip_id;
err = camera_read_reg(REG_CHIP_ID, &chip_id);
if (err != NRF_SUCCESS || chip_id != EXPECTED_CHIP_ID) {
    return NRF_ERROR_NOT_FOUND;
}

return NRF_SUCCESS;
}

/* 采集一帧图像（由image_capture_task调用，每10ms一次） */
ret_code_t camera_capture_frame(uint8_t *buf, uint16_t buf_size) {
    if (buf_size < CAMERA_IMAGE_SIZE) return NRF_ERROR_DATA_SIZE;

    /* 等待VSYNC信号（帧同步，确保从帧起始处读取） */
    uint32_t timeout = 1000;
    while (nrf_gpio_pin_read(CAMERA_VSYNC_PIN) == 0 && timeout--);
    if (timeout == 0) return NRF_ERROR_TIMEOUT;

    /* SPI DMA读取图像数据（1024字节） */
    return spi_dma_read(CAMERA_SPI_INSTANCE, buf, CAMERA_IMAGE_SIZE);
}

```

3.3 driver/pressure.c — 压力传感器驱动

压力传感器驱动读取笔尖压力ADC值，检测落笔（pen_down）和抬笔（pen_up）事件，并提供压感数值供上层使用。

ADC采样与落笔检测：

```

/* driver/pressure.c */
#define PRESSURE_ADC_CHANNEL    NRF_SAADC_INPUT_AIN0
#define PRESSURE_THRESHOLD_LOW  50    /* ADC值低于此值认为是抬笔（12位ADC，最大4095） */
#define PRESSURE_THRESHOLD_HIGH 80    /* ADC值高于此值认为是落笔 */
#define PRESSURE_OVERSAMPLE     16    /* 过采样次数，提升精度 */

/* 全局状态：当前落笔状态 */
static bool s_is_pen_down = false;

/* 采样一次压力值（含过采样平均） */
ret_code_t pressure_sample(uint8_t *pressure_normalized) {
    int32_t sum = 0;
    nrf_saadc_value_t sample;

    /* 16次过采样取均值（降低噪声） */
    for (int i = 0; i < PRESSURE_OVERSAMPLE; i++) {
        nrf_drv_saadc_sample_convert(0, &sample);
        sum += sample;
    }
    int32_t avg = sum / PRESSURE_OVERSAMPLE;

    /* 归一化到 0~255 */
    *pressure_normalized = (uint8_t)((avg * 255) / 4095);
}

```

```

/* 滞回检测（防止抖动触发误报） */
if (!s_is_pen_down && avg > PRESSURE_THRESHOLD_HIGH) {
    s_is_pen_down = true;
    /* 发布落笔事件到事件组 */
    xEventGroupSetBitsFromISR(g_pen_events, EVENT_PEN_DOWN, NULL);
} else if (s_is_pen_down && avg < PRESSURE_THRESHOLD_LOW) {
    s_is_pen_down = false;
    /* 发布抬笔事件到事件组 */
    xEventGroupSetBitsFromISR(g_pen_events, EVENT_PEN_UP, NULL);
}

return NRF_SUCCESS;
}

```

3.4 driver/battery.c — 电池电量监测驱动

```

/* driver/battery.c */
/* 电池电量（电压→百分比）查表法（基于锂电池放电曲线） */
/* 电压采用分压电路采样（2:1分压，实际电池电压=ADC读数×2×3.6V/4095） */

static const uint16_t s_voltage_table[] = {
    /* 电压（mV）：4200, 4100, 4000, 3900, 3800, 3700, 3600, 3500, 3400, 3300 */
    4200, 4100, 4000, 3900, 3800, 3700, 3600, 3500, 3400, 3300
};

static const uint8_t s_percent_table[] = {
    /* 对应百分比：100%, 90%, 80%, 70%, 60%, 40%, 20%, 10%, 5%, 0% */
    100, 90, 80, 70, 60, 40, 20, 10, 5, 0
};

uint8_t battery_get_level(void) {
    nrf_saadc_value_t adc_val;
    nrf_drv_saadc_sample_convert(1, &adc_val); /* ADC通道1：电池电压 */

    /* 计算实际电压（mV）：ADC_val * 2 * 3600mV / 4095 */
    uint32_t voltage_mv = (uint32_t)adc_val * 7200 / 4095;

    /* 查表插值计算电量百分比 */
    for (int i = 0; i < 9; i++) {
        if (voltage_mv >= s_voltage_table[i]) {
            /* 线性插值 */
            uint32_t diff_v = s_voltage_table[i] - s_voltage_table[i+1];
            uint32_t diff_p = s_percent_table[i] - s_percent_table[i+1];
            uint32_t offset = (s_voltage_table[i] - voltage_mv) * diff_p / diff_v;
            return (uint8_t)(s_percent_table[i] - offset);
        }
    }
    return 0; /* 电压过低 */
}

```

3.5 codec/dot_decoder.c — 点阵码坐标解码

点阵码解码是笔固件中最核心的算法模块，负责从摄像头采集的32×32灰度图像中识别Anoto点阵码，并解算出全球唯一的纸面坐标。

Anoto点阵码原理简述：

Anoto点阵码是一种编码纸面绝对坐标的视觉编码系统： – 纸面印刷一个由微小圆点组成的点阵图案（点间距约0.3mm，600DPI打印） – 每个圆点相对于理想网格位置有6种偏移（上/下/左/右及对角方向） – 每个偏移值编码1位或多位信息（取决于编码方案版本） – 摄像头拍摄6×6区域的点阵图案即可解算出该区域的全球唯一坐标

解码流程（定点数优化实现）：

```
/* codec/dot_decoder.c */
/* 为嵌入式MCU优化：使用Q15定点数替代浮点运算 */

typedef int16_t q15_t; /* Q15定点数：1位符号 + 15位小数 */
#define Q15_ONE      (1 << 15)

/**
 * @brief 从图像中解码点阵坐标
 * @param image      输入灰度图像（32×32字节）
 * @param x_out      输出X坐标（单位：0.01mm）
 * @param y_out      输出Y坐标（单位：0.01mm）
 * @return 0=成功，负值=解码失败（图像模糊/超出覆盖区域等）
 */
int32_t dot_decode(const uint8_t *image,
                   uint32_t *x_out, uint32_t *y_out) {

    /* Step 1: 二值化（Otsu自适应阈值，定点数版本） */
    uint8_t threshold = otsu_threshold_q15(image, 32*32);
    uint8_t binary[32*32];
    for (int i = 0; i < 32*32; i++) {
        binary[i] = (image[i] < threshold) ? 1 : 0; /* 暗点=1, 亮背景=0 */
    }

    /* Step 2: 连通域检测，提取圆点中心坐标列表 */
    dot_center_t centers[MAX_DOTS_PER_FRAME]; /* 最多36个点（6×6区域） */
    int dot_count = find_dot_centers(binary, 32, 32, centers);

    if (dot_count < 20) {
        return DOT_ERR_T00_FEW_DOTS; /* 识别到的点太少，图像质量不足 */
    }

    /* Step 3: 估计点阵网格参数（间距、旋转角度） */
    grid_params_t grid;
    if (estimate_grid_params(centers, dot_count, &grid) != 0) {
        return DOT_ERR_GRID_ESTIMATION_FAILED;
    }

    /* Step 4: 将各圆点映射到网格位置，计算偏移量 */
    uint8_t offsets[36]; /* 最多6×6=36个点的偏移编码 */
    int offset_count = calculate_dot_offsets(
        centers, dot_count, &grid, offsets);
}
```

```

/* Step 5: 从偏移序列解码坐标值（查GF2^m有限域码表） */
uint32_t x_raw, y_raw;
if (decode_position_from_offsets(offsets, offset_count,
                                &x_raw, &y_raw) != 0) {
    return DOT_ERR_POSITION_DECODE_FAILED;
}

/* Step 6: 坐标精化（亚像素级精度，利用圆点中心的插值位置） */
q15_t sub_x, sub_y;
compute_subpixel_offset(centers, dot_count, &grid, &sub_x, &sub_y);

/* Step 7: 合并整数坐标与亚像素偏移，输出0.01mm精度坐标 */
/* 1个基本坐标单位 = 0.3mm = 30个0.01mm单位 */
*x_out = x_raw * 30 + (int32_t)sub_x * 30 / Q15_ONE;
*y_out = y_raw * 30 + (int32_t)sub_y * 30 / Q15_ONE;

return 0;
}

```

解码性能指标：

指标	规格
解码延迟（正常图像）	< 3ms（@64MHz M4F，含全部步骤）
坐标精度	0.01mm（亚像素级精度）
识别率（正常书写）	≥ 99.5%（图像质量合格条件下）
抗倾斜范围	±15°笔倾斜角（DFOV 20°摄像头视野内）
最大书写速度	1.5m/s（100Hz采样，间距15mm以内不丢点）

3.6 codec/stroke_encoder.c — 笔迹数据编码打包

笔迹编码器将原始坐标帧数组打包为BLE传输格式，采用差分编码降低数据量。

差分编码原理：

由于相邻坐标帧之间的位移通常较小（书写速度有限），使用差分编码（存储坐标差值而非绝对坐标）可大幅压缩数据量： – 第一帧发送绝对坐标（4字节xy） – 后续帧发送与前一帧的差值（如差值范围在±127内，仅需1字节/维） – 典型情况下数据量减少约60%

```

/* codec/stroke_encoder.c */
uint16_t stroke_encode_packet(
    const coord_frame_t *frames,      /* 输入：坐标帧数组 */
    uint8_t frame_count,              /* 帧数 */
    uint8_t *out_buf,                 /* 输出：BLE数据包缓冲区 */
    uint16_t buf_size) {              /* 返回：实际填充字节数 */

```

```

if (frame_count == 0 || !frames || !out_buf) return 0;

uint8_t *ptr = out_buf;

/* 包头: 类型(1B) + 帧数(1B) + 时间戳(2B) */
*ptr++ = PKT_TYPE_STROKE;
*ptr++ = frame_count;
uint16_t base_ts = (uint16_t)(frames[0].seq * 10); /* 基准时间(毫秒低16位) */
memcpy(ptr, &base_ts, 2); ptr += 2;

/* 第一帧: 绝对坐标 */
memcpy(ptr, &frames[0].x, 2); ptr += 2;
memcpy(ptr, &frames[0].y, 2); ptr += 2;
*ptr++ = frames[0].pressure;
*ptr++ = frames[0].flags;

/* 后续帧: 差分编码 */
for (int i = 1; i < frame_count; i++) {
    int16_t dx = (int16_t)(frames[i].x - frames[i-1].x);
    int16_t dy = (int16_t)(frames[i].y - frames[i-1].y);

    /* 编码标志位: bit7=dx扩展(需2字节), bit6=dy扩展 */
    uint8_t enc_flags = frames[i].flags;
    if (dx < -127 || dx > 127) enc_flags |= 0x80;
    if (dy < -127 || dy > 127) enc_flags |= 0x40;
    *ptr++ = enc_flags;

    if (enc_flags & 0x80) { memcpy(ptr, &dx, 2); ptr += 2; }
    else { *ptr++ = (int8_t)dx; }

    if (enc_flags & 0x40) { memcpy(ptr, &dy, 2); ptr += 2; }
    else { *ptr++ = (int8_t)dy; }

    *ptr++ = frames[i].pressure;
}

return (uint16_t)(ptr - out_buf);
}

```

3.7 task/image_capture_task.c — 图像采集任务

图像采集任务是固件中优先级最高的任务，以100Hz（每10ms一次）定时触发，驱动摄像头采集图像并放入图像队列供坐标计算任务处理。

```

/* task/image_capture_task.c */
/* 双缓冲(Ping-Pong)设计: 采集任务写入缓冲区A时, 计算任务处理缓冲区B */
static uint8_t s_img_buf_a[CAMERA_IMAGE_SIZE];
static uint8_t s_img_buf_b[CAMERA_IMAGE_SIZE];
static bool s_use_buf_a = true; /* 当前采集写入哪个缓冲区 */

void image_capture_task(void *param) {
    TickType_t last_wake_time = xTaskGetTickCount();

```

```

const TickType_t period = pdMS_TO_TICKS(10); /* 10ms = 100Hz */

for (;;) {
    /* 精确10ms周期触发 (vTaskDelayUntil保证累计误差不漂移) */
    vTaskDelayUntil(&last_wake_time, period);

    /* 喂看门狗 (每10ms喂一次, 超时32秒触发复位) */
    nrf_drv_wdt_channel_feed(g_wdt_channel);

    /* 选择当前写入缓冲区 */
    uint8_t *write_buf = s_use_buf_a ? s_img_buf_a : s_img_buf_b;

    /* 采集一帧图像 (SPI DMA, 非阻塞) */
    ret_code_t err = camera_capture_frame(write_buf, CAMERA_IMAGE_SIZE);

    if (err == NRF_SUCCESS) {
        /* 将缓冲区指针发送到坐标计算队列 (不阻塞, 满则丢帧) */
        BaseType_t sent = xQueueSend(g_image_queue, &write_buf, 0);
        if (sent == pdTRUE) {
            /* 切换缓冲区 */
            s_use_buf_a = !s_use_buf_a;
        } else {
            /* 队列满: 坐标计算任务处理不过来, 丢弃本帧 */
            g_stat_dropped_frames++;
        }
    }
}
}

```

3.8 task/coord_calc_task.c — 坐标计算任务

坐标计算任务从图像队列读取图像帧，调用点阵码解码算法计算坐标，并将坐标数据帧写入坐标队列。

```

/* task/coord_calc_task.c */
void coord_calc_task(void *param) {
    uint8_t *img_buf;
    coord_frame_t frame;
    uint8_t seq = 0;

    for (;;) {
        /* 阻塞等待图像队列 */
        if (xQueueReceive(g_image_queue, &img_buf, portMAX_DELAY) != pdTRUE) {
            continue;
        }

        /* 采样压力值 (与图像同步, 保证时序对应) */
        uint8_t pressure;
        pressure_sample(&pressure);

        /* 调用点阵码解码算法 */
        uint32_t x, y;
        int32_t decode_result = dot_decode(img_buf, &x, &y);
    }
}

```

```

if (decode_result == 0) {
    /* 解码成功：构建坐标帧 */
    frame.x      = (uint16_t)(x & 0xFFFF);
    frame.y      = (uint16_t)(y & 0xFFFF);
    frame.pressure = pressure;
    frame.seq     = seq++;
    frame.flags   = (pressure < PRESSURE_THRESHOLD_LOW) ? FLAG_PEN_UP : 0;

    /* 低电量标志 */
    if (battery_get_level() <= BATTERY_LOW_THRESHOLD) {
        frame.flags |= FLAG_BATTERY_LOW;
    }

    /* 写入坐标队列（BLE发送任务消费） */
    xQueueSend(g_coord_queue, &frame, 0);

    /* 如果BLE未连接，写入离线Flash缓存 */
    if (!ble_is_connected()) {
        offline_cache_write(&frame);
    }
} else {
    /* 解码失败（图像模糊/笔尖抬起）：发送纯压力帧 */
    frame.x      = 0;
    frame.y      = 0;
    frame.pressure = pressure;
    frame.seq     = seq++;
    frame.flags   = FLAG_PEN_UP | FLAG_NO_POSITION;

    if (pressure < PRESSURE_THRESHOLD_LOW) {
        /* 确认抬笔，向BLE发送抬笔事件 */
        xQueueSend(g_coord_queue, &frame, 0);
    }
}
}
}
}

```

3.9 task/ble_send_task.c — BLE数据发送任务

BLE发送任务从坐标队列读取坐标帧，积累一定数量后打包编码，通过BLE Notify方式发送至已连接的设备。

```

/* task/ble_send_task.c */
#define BLE_NOTIFY_INTERVAL_MS 20 /* 每20ms发送一次（50Hz发送，100Hz采样2帧/包） */
#define MAX_FRAMES_PER_PKT 34 /* 每包最多34帧（适配BLE MTU=247字节） */

void ble_send_task(void *param) {
    coord_frame_t batch[MAX_FRAMES_PER_PKT];
    uint8_t batch_count = 0;
    TickType_t last_send_time = xTaskGetTickCount();

    for (;;) {

```

```

coord_frame_t frame;
/* 等待坐标帧（最多等到下次发送时刻） */
TickType_t wait_time = pdMS_TO_TICKS(BLE_NOTIFY_INTERVAL_MS);

if (xQueueReceive(g_coord_queue, &frame, wait_time) == pdTRUE) {
    batch[batch_count++] = frame;
}

/* 达到发送时间或包满 → 打包发送 */
bool time_to_send = (xTaskGetTickCount() - last_send_time) >=
    pdMS_TO_TICKS(BLE_NOTIFY_INTERVAL_MS);

if ((batch_count > 0) && (time_to_send || batch_count >= MAX_FRAMES_PER_PKT)) {

    if (ble_is_connected() && ble_notify_enabled()) {
        /* 编码为BLE数据包 */
        uint8_t pkt_buf[BLE_MAX_MTU];
        uint16_t pkt_len = stroke_encode_packet(
            batch, batch_count, pkt_buf, sizeof(pkt_buf));

        /* 发送BLE Notify */
        uint32_t err = ble_nus_data_send(
            &m_ble_conn_handle, pkt_buf, &pkt_len);

        if (err == NRF_SUCCESS) {
            g_stat_ble_packets_sent++;
            g_stat_ble_bytes_sent += pkt_len;
        } else if (err == NRF_ERROR_RESOURCES) {
            /* BLE发送缓冲区满，重试（最多3次） */
            vTaskDelay(pdMS_TO_TICKS(5));
            ble_nus_data_send(&m_ble_conn_handle, pkt_buf, &pkt_len);
        }
    }

    batch_count = 0;
    last_send_time = xTaskGetTickCount();
}
}
}

```

3.10 task/power_task.c — 电源管理任务

电源管理任务以1Hz频率运行，负责电量采样、充电状态监测和功耗模式转换。

功耗模式转换策略：

```

/* task/power_task.c */
/* 电源状态机定义 */
typedef enum {
    POWER_STATE_ACTIVE,      /* 活跃：BLE连接且持续书写 */
    POWER_STATE_IDLE,        /* 空闲：BLE连接但无书写（持续5秒） */
    POWER_STATE_SLEEP,       /* 休眠：无BLE连接（保留广播，降低摄像头帧率） */
    POWER_STATE_DEEP_SLEEP   /* 深度休眠：无书写且无BLE超过3分钟 */
} power_state_t;

```

```

static power_state_t s_power_state = POWER_STATE_SLEEP;
static uint32_t s_idle_seconds = 0;    /* 空闲计时 */
static uint32_t s_no_write_seconds = 0; /* 无书写计时 */

void power_task(void *param) {
    TickType_t last_wake = xTaskGetTickCount();

    for (;;) {
        vTaskDelayUntil(&last_wake, pdMS_TO_TICKS(1000)); /* 1Hz */

        /* 1. 采样电池电量 */
        uint8_t level = battery_get_level();
        g_battery_level = level;

        /* 低电量告警 (≤15%: 慢闪红灯; ≤5%: 发BLE通知后关机) */
        if (level <= 5) {
            ble_send_battery_notify(level);
            vTaskDelay(pdMS_TO_TICKS(500));
            power_system_off(); /* 进入System OFF (0.2μA) */
        } else if (level <= 15) {
            xEventGroupSetBits(g_led_events, LED_EVENT_LOW_BATTERY);
        }

        /* 2. 检查充电状态 (通过I2C读充电IC状态寄存器) */
        bool is_charging = charger_is_charging();
        if (is_charging != g_is_charging) {
            g_is_charging = is_charging;
            xEventGroupSetBits(g_led_events, LED_EVENT_CHARGE_CHANGED);
        }

        /* 3. 功耗状态机转换 */
        bool pen_is_writing = (g_last_coord_seq_changed_within_1s);
        bool ble_connected = ble_is_connected();

        power_state_t new_state = s_power_state;

        switch (s_power_state) {
            case POWER_STATE_ACTIVE:
                if (!pen_is_writing) {
                    s_idle_seconds++;
                    if (s_idle_seconds > 5) new_state = POWER_STATE_IDLE;
                } else {
                    s_idle_seconds = 0;
                }
                break;

            case POWER_STATE_IDLE:
                if (pen_is_writing) {
                    new_state = POWER_STATE_ACTIVE;
                    s_idle_seconds = 0;
                } else if (!ble_connected) {
                    new_state = POWER_STATE_SLEEP;
                }
                break;

            case POWER_STATE_SLEEP:

```

```

        if (ble_connected && pen_is_writing) {
            new_state = POWER_STATE_ACTIVE;
            s_no_write_seconds = 0;
        } else {
            s_no_write_seconds++;
            if (s_no_write_seconds > 180) { /* 3分钟无操作 */
                new_state = POWER_STATE_DEEP_SLEEP;
            }
        }
        break;

    case POWER_STATE_DEEP_SLEEP:
        /* 按下笔帽按键唤醒 (SENSE引脚中断) */
        break;
}

/* 状态转换处理 */
if (new_state != s_power_state) {
    power_apply_state(new_state);
    s_power_state = new_state;
}
}
}

```

3.11 task/led_task.c — LED状态指示任务

LED任务控制RGB三色LED，根据系统状态显示不同颜色和动效，提供直观的用户反馈。

LED状态对应表：

状态	LED颜色	动效	说明
开机/广播中	蓝色	慢闪（1Hz）	等待蓝牙连接
配对请求	白色	快闪（4Hz）	等待用户确认配对
已连接	蓝色	常亮	BLE连接正常
书写中	蓝色	呼吸灯（2s周期）	检测到书写动作
充电中	红色	慢闪（0.5Hz）	USB充电中
充电完成	绿色	常亮	电量100%
低电量（≤15%）	红色	慢闪（1Hz）	需要充电
极低电量（≤5%）	红色	快闪（4Hz）	即将自动关机
OTA升级中	黄色	快闪	固件升级进行中
OTA成功	绿色	闪3次后熄灭	升级成功，即将重启

状态	LED颜色	动效	说明
OTA失败	红色	闪3次后恢复正常	升级失败，保持原版本

3.12 task/ota_task.c — OTA固件升级任务

OTA任务实现通过BLE DFU协议接收固件包，验证后写入B分区并触发系统重启以完成升级。

OTA状态机：

```
/* task/ota_task.c */
typedef enum {
    OTA_STATE_IDLE = 0,
    OTA_STATE_INIT,          /* 初始化：清除B分区，准备接收 */
    OTA_STATE_RECEIVING,     /* 接收固件分块数据 */
    OTA_STATE_VERIFYING,     /* 校验CRC32 + RSA签名 */
    OTA_STATE_WRITING,       /* 写入Flash B分区 */
    OTA_STATE_DONE,          /* 完成，等待重启确认 */
    OTA_STATE_ERROR          /* 错误，放弃升级 */
} ota_state_t;

/* OTA完成后的处理 */
static void ota_finalize_upgrade(void) {
    /* 1. 在NVS中设置"切换到B分区"标志 */
    nvs_write_u8(NVS_KEY_OTA_FLAG, OTA_SWITCH_TO_B);

    /* 2. 发送OTA完成通知 (BLE Indicate) */
    ble_dfu_send_status(DFU_STATUS_SUCCESS, 100);

    /* 3. 延迟1秒等待BLE数据发送完成，然后重启 */
    vTaskDelay(pdMS_TO_TICKS(1000));

    /* 4. 触发软件复位 (AIRCR.SYSRESETREQ) */
    sd_nvic_SystemReset();
}
```

3.13 cache/offline_cache.c — 离线数据缓存

离线缓存模块管理外部SPI Flash的坐标数据缓存，在BLE未连接时保存书写数据，连接后自动同步。

FIFO环形缓存设计：

外部Flash 4MB，划分为：

- 4096个扇区（每扇区1KB）
- 采用双指针FIFO设计：
write_ptr：下一次写入的扇区位置
read_ptr：下一次读取的扇区位置

写入：每组数据（group_header + N×coord_frame）对齐到扇区边界
write_ptr + 1 == read_ptr（满）时：停止写入，丢弃最新数据
读取：连接后顺序读取 read_ptr 到 write_ptr 之间的所有数据
清空：成功同步后，read_ptr = write_ptr（逻辑清空，无需实际擦除）
磨损均衡：每2000次写入后将头部扇区向后移动，均匀磨损各扇区

3.14 power/power_manager.c — 低功耗状态机

电源管理器根据功耗状态对各硬件模块进行电源控制，在不影响功能的前提下最大化续航。

各功耗状态的硬件配置：

功耗状态	摄像头	ADC采样率	BLE广播间隔	CPU速度	功耗估算
ACTIVE（活跃）	100fps	100Hz	连接态（CI 15ms）	64MHz	~15mA
IDLE（空闲）	10fps	10Hz	连接态（CI 200ms）	16MHz	~5mA
SLEEP（休眠）	关闭	1Hz	1s广播间隔	4MHz	~0.5mA
DEEP_SLEEP（深睡）	关闭	关闭	关闭广播	停止（待中断唤醒）	~2μA

第四章 操作流程与使用步骤

4.1 点阵笔开机流程

用户操作：按下笔帽末端开关键（长按1.5秒开机）

- └ 固件检测按键中断（GPIO SENSE唤醒）
- └ 执行 main() 初始化流程（约300ms）
 - └ 硬件自检（Flash R/W测试、摄像头ID校验）
 - └ 自检失败→红色快闪3次提示硬件故障
- └ 读取NVS配对记录
 - └ 有配对记录→优先尝试定向广播（针对已配对设备）
 - └ 无配对记录→开启无向广播（等待新设备配对）
- └ 开始广播
 - └ LED：蓝色慢闪（等待连接）
- └ 等待BLE连接（或超时60秒后进入SLEEP模式节省电量）

4.2 蓝牙配对与连接流程

新设备首次配对：



已配对设备重连（自动）：

固件在广播时携带已配对设备的地址（定向广播）。已配对的终端APP在扫描时发现该广播后，自动发起重连请求，固件验证LTK后恢复加密连接，全程无需用户干预（约3秒完成重连）。

4.3 书写采集与数据传输流程

学生书写流程（实时传输模式）：

1. 用户持笔书写于点阵纸上
|
2. 笔尖接触纸面（压力传感器 $\text{pressure} > \text{阈值}$ ）
→ 触发 `EVENT_PEN_DOWN`
|
3. `image_capture_task`: 100Hz连续采集摄像头图像
→ 写入双缓冲图像队列
|
4. `coord_calc_task`: 实时解码每帧图像
→ 输出精确坐标 $(x, y, \text{pressure}, \text{flags})$
→ 写入坐标队列
|
5. `ble_send_task`: 每20ms积累2帧，打包差分编码
→ BLE Notify发送（BLE包约20字节）
|
6. 笔尖离开纸面（ $\text{pressure} < \text{阈值}$ ）

- 触发 EVENT_PEN_UP
- 发送抬笔标志帧 (flags |= FLAG_PEN_UP)

7. 终端APP (网关/黑板/手机) 收到坐标数据
 - 转发至云端或算力盒进行AI识别

4.4 离线书写与数据同步流程

离线书写 (无BLE连接时):

1. 笔处于SLEEP模式 (无连接)
 - |
2. 用户开始书写 → 摄像头采集 → 坐标解码
 - |
3. coord_calc_task检测到BLE未连接
 - offline_cache_write(frame) 写入外部Flash
 - |
4. 持续书写, 每组数据带时间戳写入缓冲
 - (缓存满4MB=约10万点=约10页书写时, 停止缓存, 仅更新时间戳)
 - |

离线数据同步 (重新建立BLE连接时):

5. BLE连接建立, ble_evt_handler触发 EVT_CONNECTED
 - |
6. ble_send_task 检测到有离线数据 (offline_cache_get_count() > 0)
 - 先向对端发送"离线数据同步开始"通知
 - 切换数据包类型为 PKT_TYPE_OFFLINE_SYNC
 - |
7. 顺序读取Flash缓存, 按组发送 (每组含时间戳信息)
 - 使用 BLE GATT Indicate (需要ACK确认, 保证可靠传输)
 - |
8. 所有数据发送完成
 - 接收方确认 (ACK)
 - offline_cache_clear() 清空Flash缓存 (移动read_ptr)
 - 切换回实时传输模式

4.5 充电与电量管理

充电状态机:

USB接入 → 充电IC检测到电源

- |
- 充电开始: LED红色慢闪
 - 电量 < 80%: 500mA快充
 - 电量 80-100%: 涓流充电
- |
- 充电完成 (电量100%): LED绿色常亮
- |
- USB拔出: LED恢复蓝色状态指示

电量提示规则 (写入BLE Notify主动上报):

- 电量每变化1%时: 更新 Battery Level Characteristic
- 电量降至50%: 主动Notify一次 (提醒用户关注电量)

- 电量降至15%: LED红色慢闪 + 主动Notify
- 电量降至5%: LED红色快闪 + Notify + 30秒后自动关机

4.6 OTA固件升级流程

通过终端APP进行OTA（用户视角）：

1. 厂商将新固件包（.zip，含.bin + RSA签名文件）上传至云端
2. 终端APP（手机/PC）检测到新版本，提示用户升级
3. 用户点击"立即升级"，APP通过BLE与点阵笔建立DFU连接
4. APP自动完成固件下载和传输（约2-5分钟，取决于BLE信号强度）
5. LED变为黄色快闪（升级进行中），用户保持笔与APP蓝牙距离
6. 升级完成：LED绿色闪3次，笔自动重启（约3秒）
7. 重启后连接，APP显示"固件已更新至vX.X.X"

升级异常处理：

异常情况	处理方式
传输中断（BLE断开）	支持断点续传（记录已接收分块序号），重连后继续
CRC校验失败	放弃本次升级，保留A分区，LED红色闪3次
RSA签名验证失败	拒绝安装，视为非法固件，触发安全告警
B分区启动失败	Bootloader自动回滚至A分区，版本不变
升级中电量不足	电量<20%时拒绝启动OTA，提示先充电

4.7 出厂校准与测试流程

生产烧录流程：

- 生产线操作步骤：
1. 连接J-Link调试器（SWD接口）
 2. 烧录 Bootloader（0x27000）
 3. 烧录 SoftDevice（0x00000000）
 4. 烧录 App A 固件（0x30000）
 5. 写入 NVS 出厂信息：
 - 设备序列号（唯一，扫描二维码写入）
 - 硬件版本号
 - 摄像头校准参数（每支笔独立校准）
 6. 启用 APPROTECT（Flash读保护）
 7. 自动化测试：
 - BLE广播检测（天线测试）
 - 摄像头采集测试（点阵纸图像解码验证）

- 压力传感器量程测试
- 电池电量校准

4.8 常见问题处理

问题现象	可能原因	处理方法
笔无法开机	电量耗尽	充电至少5分钟后再开机
BLE无法发现笔	广播超时进入SLEEP	长按笔帽开关1.5秒重新开机
书写坐标飘移	摄像头焦距偏移	联系售后，执行校准流程
离线数据同步慢	缓存数据量大	保持BLE连接，等待同步完成（约每1000点需10秒）
OTA升级失败	固件包版本低于当前	检查APP中固件版本，使用正确的升级包
LED不亮	LED驱动故障	功能不受影响，联系售后检测

第五章 与源代码的对应关系

5.1 模块名称与源代码文件对应表

文档章节	源代码文件	语言	说明
主程序与RTOS启动	main.c	C	系统初始化、任务创建、BLE协议栈配置
点阵摄像头驱动	driver/camera.c	C	SPI摄像头图像采集驱动
压力传感器驱动	driver/pressure.c	C	ADC压力采样，落笔/抬笔事件检测
电池电量检测	driver/battery.c	C	电池电压ADC采样，查表法计算电量
外部Flash驱动	driver/flash.c	C	SPI NOR Flash读写驱动，磨损均衡
LED驱动	driver/led.c	C	RGB LED PWM驱动，动效控制
点阵码解码	codec/dot_decoder.c	C	Anoto点阵码识别与坐标解算（定点数实现）
笔迹数据编码	codec/stroke_encoder.c	C	差分编码，BLE数据包打包
图像采集任务	task/image_capture_task.c	C	100Hz定时摄像头采集RTOS任务
坐标计算任务	task/coord_calc_task.c	C	调用解码算法，输出坐标帧

文档章节	源代码文件	语言	说明
BLE数据发送任务	task/ble_send_task.c	C	坐标打包编码，BLE Notify发送
电源管理任务	task/power_task.c	C	电量采样，功耗状态机管理
LED状态指示任务	task/led_task.c	C	事件驱动LED动效控制
OTA固件升级任务	task/ota_task.c	C	BLE DFU协议，固件包接收校验写入
离线数据缓存	cache/offline_cache.c	C	外部Flash FIFO缓存管理
低功耗管理	power/power_manager.c	C	四级功耗状态切换，硬件电源控制
配对管理	ble/peer_manager.c	C	BLE配对记录NVS存取管理
BLE事件处理	ble/ble_evt_handler.c	C	SoftDevice BLE事件回调分发
GATT Service实现	ble/gatt_services.c	C	自定义GATT Service注册与数据处理
中断向量表	startup/startup_nrf52840.s	ASM	芯片启动文件，中断向量表
链接脚本	ld/nrf52840_app.ld	LD	Flash/RAM分区分配

5.2 核心函数说明

函数名	所在文件	功能说明
main()	main.c	固件启动入口，硬件初始化与RTOS任务创建
camera_init()	driver/camera.c	摄像头SPI初始化，寄存器配置
camera_capture_frame()	driver/camera.c	采集一帧图像（SPI DMA读取）
pressure_sample()	driver/pressure.c	采样一次压力值，检测落笔/抬笔事件
battery_get_level()	driver/battery.c	读取电池电量百分比
dot_decode()	codec/dot_decoder.c	核心算法：从图像解算点阵坐标

函数名	所在文件	功能说明
otsu_threshold_q15()	codec/dot_decoder.c	定点数Otsu自适应二值化
stroke_encode_packet()	codec/stroke_encoder.c	差分编码打包为BLE数据包
image_capture_task()	task/image_capture_task.c	100Hz图像采集RTOS任务
coord_calc_task()	task/coord_calc_task.c	坐标解算RTOS任务
ble_send_task()	task/ble_send_task.c	BLE数据发送RTOS任务
power_task()	task/power_task.c	电源监控与状态机管理
power_apply_state()	power/power_manager.c	执行功耗状态切换（调整各外设电源）
offline_cache_write()	cache/offline_cache.c	写入一帧坐标到Flash离线缓存
offline_cache_sync()	cache/offline_cache.c	将离线缓存数据读出供BLE发送
ota_finalize_upgrade()	task/ota_task.c	OTA完成后写标志并触发重启
enable_flash_protection()	main.c	启用APPROTECT Flash读保护（生产版本）

5.3 寄存器与GATT特征定义

自定义摄像头寄存器（camera_regs.h）：

宏定义	寄存器地址	说明
REG_RESET	0x12	软件复位（写0x80触发）
REG_CHIP_ID	0x0A	芯片ID（只读，期望值0xAB）
REG_EXPOSURE_H	0x10	曝光时间高字节
REG_EXPOSURE_L	0x11	曝光时间低字节
REG_GAIN_CTRL	0x13	增益控制（AGC设置）
REG_CLKDIV	0x11	时钟分频（帧率控制）
REG_FORMAT	0x12	输出格式（0=灰度，1=Bayer）

BLE Characteristic UUID映射（gatt_services.h）：

宏定义	UUID值	属性	说明
UUID_STROKE_DATA_CHAR	0xFFFF1	Notify	笔迹坐标数据
UUID_PEN_CONTROL_CHAR	0xFFFF2	Write	控制指令接收
UUID_BATTERY_CHAR	0xFFFF3	Read/Notify	电池电量
UUID_DEVICE_SERIAL_CHAR	0xFEE1	Read	设备序列号
UUID_FW_VERSION_CHAR	0xFEE2	Read	固件版本
UUID_HW_VERSION_CHAR	0xFEE3	Read	硬件版本
UUID_CALIBRATION_CHAR	0xFEE4	Read/Write	摄像头校准参数
UUID_DFU_CONTROL_CHAR	0xFE1	Write/Indicate	OTA控制
UUID_DFU_PACKET_CHAR	0xFE2	Write Without Response	OTA数据包
UUID_DFU_STATUS_CHAR	0xFE3	Indicate	OTA状态上报

附录A BLE GATT服务定义表

A.1 Writtech Pen Data Service (Primary Service)

- Service UUID: 0000FFF0-0000-1000-8000-00805F9B34FB

Characteristic	UUID	Properties	Value Length	Description
Stroke Data	0000FFF1-...	Notify	最大247字节	差分编码坐标数据包
Pen Control	0000FFF2-...	Write	2字节	byte[0]=命令类型, byte[1]=参数
Battery Level	0000FFF3-...	Read, Notify	1字节	电量百分比 (0~100)

A.2 Writtech Device Info Service (Primary Service)

- Service UUID: 0000FEE0-0000-1000-8000-00805F9B34FB

Characteristic	UUID	Properties	Value Length	Description
Device Serial	0000FEE1-...	Read	16字节	ASCII序列号
Firmware Version	0000FEE2-...	Read	8字节	版本字符串（如"V1.0.0\0\0"）
Hardware Version	0000FEE3-...	Read	8字节	硬件版本字符串
Calibration	0000FEE4-...	Read, Write	32字节	摄像头标定参数（结构体序列化）

附录B 硬件外设寄存器说明

B.1 SPI摄像头接线

nRF52840 引脚	摄像头引脚	说明
P0.03	CS	片选（低有效）
P0.04	MOSI	主发从收（配置数据）
P0.05	MISO	主收从发（图像数据）
P0.06	SCK	SPI时钟（最高8MHz）
P0.07	PWDN	电源使能（高有效）
P0.08	VSYNC	帧同步信号（输入）

B.2 外部Flash（W25Q32）接线

nRF52840 引脚	Flash引脚	说明
P1.00	CS	片选（低有效）
P1.01	MOSI	数据输入
P1.02	MISO	数据输出
P1.03	CLK	时钟（最高50MHz）
P1.04	WP	写保护（固定高电平）

nRF52840 引脚	Flash引脚	说明
P1.05	HOLD	保持（固定高电平）

附录C 术语表

术语	说明
MCU	Microcontroller Unit，微控制单元（点阵笔主控芯片）
RTOS	Real-Time Operating System，实时操作系统
FreeRTOS	开源实时操作系统，广泛用于嵌入式系统
SoftDevice	Nordic Semiconductor的BLE协议栈软件（运行于MCU低地址空间）
BLE GATT	Generic Attribute Profile，蓝牙通用属性规范（BLE应用层协议）
Notify	BLE GATT的一种数据传输方式（无ACK，高速）
Indicate	BLE GATT的另一种数据传输方式（有ACK，可靠）
DFU	Device Firmware Update，设备固件更新（OTA的BLE标准实现）
ADC	Analog-to-Digital Converter，模数转换器
SPI	Serial Peripheral Interface，串行外设接口
NVS	Non-Volatile Storage，非易失性存储（Flash存储区域）
LTK	Long-Term Key，BLE长期密钥（配对后用于重连加密）
APPROTECT	Access Port Protection，Flash访问保护（防调试器读取）
ECDH	椭圆曲线Diffie-Hellman密钥交换（BLE Secure Connections中使用）
Anoto	一种点阵码编码体系，用于纸笔数字化（Anoto AB公司专利）
DFOV	Diagonal Field of View，对角视场角（摄像头参数）
Q15	定点数格式：1位符号位 + 15位小数位（嵌入式优化浮点替代方案）
MTU	Maximum Transmission Unit，BLE最大传输单元（默认23字节，可协商至247字节）

附录D 版本历史

版本	日期	变更说明	编制人
V0.3 Alpha	2025-06-01	基础BLE连接，坐标采集与发送	固件研发团队
V0.7 Beta	2025-09-10	离线缓存（Flash FIFO），点阵码解码精度优化（引入亚像素插值）	固件研发团队
V0.9 RC	2025-11-20	OTA升级（BLE DFU），低功耗优化（四级功耗状态机，续航延长30%）	固件研发团队
V1.0	2026-02-14	正式版：RSA签名OTA、压力传感器滞回优化、LED动效、Flash读保护	固件研发团队

文档编制：深圳自然写科技有限公司 硬件/固件研发部

文档版本：V1.0

最后更新：2026年2月14日

版权所有 © 2026 深圳自然写科技有限公司

附录E 核心算法与驱动详述

E.1 点阵码解码算法

自然写智能点阵笔采用专利点阵纸技术，通过CMOS图像传感器拍摄纸面点阵图案，由固件实时解码为绝对坐标。

E.1.1 点阵图像采集与预处理

```
/* codec/dot_codec.c - 点阵码解码主流程 */

#include "dot_codec.h"
#include "camera_driver.h"
#include "math_utils.h"

/* 点阵图像参数 */
#define IMG_WIDTH      128
#define IMG_HEIGHT     128
#define DOT_THRESHOLD  80    /* 二值化阈值 (0-255) */
#define MIN_DOT_AREA   4     /* 最小点面积 (像素, 过滤噪声) */
#define MAX_DOT_AREA   25    /* 最大点面积 (像素, 过滤粘连) */
#define EXPECTED_DOTS  64    /* 每帧期望识别到的点数 */

/* 解码结果结构 */
typedef struct {
```

```

    float abs_x;          /* 绝对X坐标 (mm), 精度0.01mm */
    float abs_y;          /* 绝对Y坐标 (mm), 精度0.01mm */
    float angle;          /* 笔的旋转角度 (度) */
    uint8_t page_id;      /* 当前纸张页码 */
    uint8_t quality;      /* 解码质量评分 (0-100) */
} DotDecodeResult;

/**
 * @brief 点阵图像二值化 (Otsu自适应阈值)
 * @param[in] src 原始灰度图 (128×128)
 * @param[out] dst 二值图 (0或255)
 */
static void binarize_otsu(const uint8_t *src, uint8_t *dst) {
    uint32_t hist[256] = {0};
    uint32_t total = IMG_WIDTH * IMG_HEIGHT;

    /* 统计直方图 */
    for (uint32_t i = 0; i < total; i++) {
        hist[src[i]]++;
    }

    /* Otsu方法求最优阈值 */
    uint32_t sum = 0;
    for (int i = 0; i < 256; i++) sum += i * hist[i];

    uint32_t sumB = 0, wB = 0, wF = 0;
    float maxVariance = 0.0f;
    uint8_t threshold = DOT_THRESHOLD;

    for (int t = 0; t < 256; t++) {
        wB += hist[t];
        if (wB == 0) continue;
        wF = total - wB;
        if (wF == 0) break;

        sumB += t * hist[t];
        float mB = (float)sumB / wB;
        float mF = (float)(sum - sumB) / wF;
        float variance = (float)wB * wF * (mB - mF) * (mB - mF);

        if (variance > maxVariance) {
            maxVariance = variance;
            threshold = (uint8_t)t;
        }
    }

    /* 应用阈值 */
    for (uint32_t i = 0; i < total; i++) {
        dst[i] = (src[i] > threshold) ? 0 : 255; /* 点为暗色, 背景为亮色 */
    }
}

/**
 * @brief 连通区域标记 (4-连通BFS), 提取各点的质心坐标
 * @param[in] binary 二值图
 * @param[out] dots 检测到的点质心数组
 * @param[out] dot_count 检测到的点数量

```

```

* @return 0成功, -1失败
*/
static int extract_dot_centroids(const uint8_t *binary,
    float *dots_x, float *dots_y, int *dot_count) {
    static uint8_t label_map[IMG_WIDTH * IMG_HEIGHT];
    memset(label_map, 0, sizeof(label_map));
    *dot_count = 0;

    /* 简化BFS标记 (嵌入式环境优化版本, 避免递归) */
    static uint16_t queue[MAX_DOT_AREA * 4];
    int label = 1;

    for (int y = 1; y < IMG_HEIGHT - 1; y++) {
        for (int x = 1; x < IMG_WIDTH - 1; x++) {
            int idx = y * IMG_WIDTH + x;
            if (binary[idx] == 0 || label_map[idx] != 0) continue;

            /* BFS flood fill */
            int head = 0, tail = 0;
            int area = 0;
            float sum_x = 0, sum_y = 0;
            queue[tail++] = (uint16_t)(y * IMG_WIDTH + x);
            label_map[idx] = label;

            while (head < tail && area < MAX_DOT_AREA * 2) {
                uint16_t cur = queue[head++];
                int cy = cur / IMG_WIDTH;
                int cx = cur % IMG_WIDTH;
                sum_x += cx;
                sum_y += cy;
                area++;

                /* 检查4邻居 */
                int neighbors[4] = {
                    (cy-1)*IMG_WIDTH+cx, (cy+1)*IMG_WIDTH+cx,
                    cy*IMG_WIDTH+(cx-1), cy*IMG_WIDTH+(cx+1)
                };
                for (int n = 0; n < 4; n++) {
                    int ni = neighbors[n];
                    if (ni >= 0 && ni < IMG_WIDTH*IMG_HEIGHT
                        && binary[ni] == 0 && label_map[ni] == 0) {
                        label_map[ni] = label;
                        queue[tail++] = (uint16_t)ni;
                    }
                }
            }

            /* 过滤面积不合理的区域 */
            if (area >= MIN_DOT_AREA && area <= MAX_DOT_AREA
                && *dot_count < EXPECTED_DOTS) {
                dots_x[*dot_count] = sum_x / area;
                dots_y[*dot_count] = sum_y / area;
                (*dot_count)++;
            }
            label++;
        }
    }
}

```

```

    return (*dot_count >= 16) ? 0 : -1; /* 少于16个点则解码失败 */
}

/**
 * @brief 点阵主解码函数
 * @param[in] raw_image CMOS原始灰度图像数据
 * @param[out] result 解码结果
 * @return 0成功, -1失败
 */
int dot_codec_decode(const uint8_t *raw_image, DotDecodeResult *result) {
    static uint8_t binary_buf[IMG_WIDTH * IMG_HEIGHT];
    static float dots_x[EXPECTED_DOTS];
    static float dots_y[EXPECTED_DOTS];
    int dot_count = 0;

    /* Step 1: 二值化 */
    binarize_otsu(raw_image, binary_buf);

    /* Step 2: 提取点质心 */
    if (extract_dot_centroids(binary_buf, dots_x, dots_y, &dot_count) != 0) {
        result->quality = 0;
        return -1;
    }

    /* Step 3: 点阵网格拟合 (最小二乘法求仿射变换参数) */
    float angle, scale, offset_x, offset_y;
    if (fit_dot_grid(dots_x, dots_y, dot_count,
        &angle, &scale, &offset_x, &offset_y) != 0) {
        result->quality = 10;
        return -1;
    }

    /* Step 4: 从网格偏移量中解码Anoto绝对坐标 */
    uint32_t abs_x_raw, abs_y_raw;
    uint8_t page_id;
    if (anoto_decode_position(dots_x, dots_y, dot_count,
        angle, &abs_x_raw, &abs_y_raw, &page_id) != 0) {
        result->quality = 30;
        return -1;
    }

    result->abs_x = abs_x_raw * 0.01f; /* 转换为mm */
    result->abs_y = abs_y_raw * 0.01f;
    result->angle = angle;
    result->page_id = page_id;
    result->quality = (uint8_t)(50 + dot_count); /* 粗略质量评分 */
    if (result->quality > 100) result->quality = 100;

    return 0;
}

```

E.2 压力传感器驱动与滞回补偿

```

/* driver/pressure_driver.c - 压力传感器驱动（带滞回补偿） */

#include "pressure_driver.h"
#include "adc_driver.h"

/* 压力ADC参数 */
#define PRESSURE_ADC_CHANNEL    1
#define PRESSURE_ADC_BITS      12      /* 12位ADC: 0~4095 */
#define PRESSURE_MIN_RAW       150     /* 最小有效ADC值（笔尖接触压力阈值） */
#define PRESSURE_MAX_RAW       3800    /* 最大ADC值（最大压力） */

/* 滞回补偿参数（防止轻微抖动导致反复触发抬笔/落笔） */
#define HYSTERESIS_LOW         180     /* 落笔阈值（下降） */
#define HYSTERESIS_HIGH        220     /* 抬笔阈值（上升） */

/* IIR低通滤波器系数（ $\alpha=0.3$ ，截止频率约48Hz@200Hz采样率） */
#define FILTER_ALPHA_FP        0.3f

static uint16_t g_pressure_filtered = 0;
static bool g_pen_down = false;

/**
 * @brief 读取并滤波压力值
 * @return 归一化压力值 [0, 255]
 */
uint8_t pressure_read_normalized(void) {
    uint16_t raw = adc_read(PRESSURE_ADC_CHANNEL);

    /* IIR一阶低通滤波 */
    g_pressure_filtered = (uint16_t)(
        FILTER_ALPHA_FP * raw + (1.0f - FILTER_ALPHA_FP) * g_pressure_filtered
    );

    /* 线性归一化到 [0, 255] */
    if (g_pressure_filtered <= PRESSURE_MIN_RAW) return 0;
    if (g_pressure_filtered >= PRESSURE_MAX_RAW) return 255;

    return (uint8_t)(
        (uint32_t)(g_pressure_filtered - PRESSURE_MIN_RAW) * 255
        / (PRESSURE_MAX_RAW - PRESSURE_MIN_RAW)
    );
}

/**
 * @brief 获取笔尖状态（带滞回，防抖）
 * @return true=笔尖接触纸面（落笔），false=笔尖离开（抬笔）
 */
bool pressure_is_pen_down(void) {
    uint16_t raw = g_pressure_filtered;

    if (!g_pen_down && raw > HYSTERESIS_HIGH) {
        g_pen_down = true; /* 落笔 */
    } else if (g_pen_down && raw < HYSTERESIS_LOW) {
        g_pen_down = false; /* 抬笔 */
    }

    /* 在 HYSTERESIS_LOW ~ HYSTERESIS_HIGH 之间保持上一状态（滞回区） */
}

```



```
return g_pen_down;
}
```

E.3 BLE GATT Service/Characteristic完整定义

智能点阵笔的BLE GATT服务定义如下，遵循Nordic UART Service规范扩展：

服务/特征	UUID	属性	说明
Writech Pen Service	6E400001-...	-	主服务
↳ Ink Data Char	6E400002-...	Notify	笔迹数据推送（10字节/点）
↳ Control Char	6E400003-...	Write	控制命令（开始/停止/配置）
↳ Status Char	6E400004-...	Read/Notify	设备状态（电量/连接/错误）
↳ OTA Data Char	6E400005-...	Write	OTA固件数据传输
↳ OTA Control Char	6E400006-...	Write/Notify	OTA控制与进度反馈
Device Info Service	0x180A	-	标准设备信息服务
↳ Manufacturer	0x2A29	Read	"Writech Technology"
↳ Model Number	0x2A24	Read	"WritechPen-M1"
↳ Firmware Version	0x2A26	Read	当前固件版本字符串
Battery Service	0x180F	-	标准电池服务
↳ Battery Level	0x2A19	Read/Notify	电池电量（0-100%）

E.3.1 笔迹数据包二进制格式

每个笔迹数据通知包（Notify包）包含1至N个笔迹点（受BLE MTU限制，默认MTU=247字节，每包最多23个点）：

每个点：10字节

X[2B]	Y[2B]	P[1B]	Timestamp[4B]	F[1B]
-------	-------	-------	---------------	-------

X: uint16_be, 归一化坐标×65535，对应纸面X轴

Y: uint16_be, 归一化坐标×65535，对应纸面Y轴

P: uint8, 压力×255

Timestamp: uint32_be, 毫秒时间戳（设备本地时钟）

F: 标志位

Bit0: 1=抬笔（笔画结束），0=落笔

Bit1: 1=笔迹质量低（解码置信度<50%）

Bit2: 1=缓存数据（离线恢复发送）
Bit3-7: 保留

E.4 四级功耗状态机

```
/* power/power_manager.c - 四级功耗管理状态机 */

typedef enum {
    POWER_STATE_ACTIVE    = 0, /* 活跃: 书写中, 全速运行 */
    POWER_STATE_IDLE      = 1, /* 空闲: 静止10s, 降低采样率 */
    POWER_STATE_SLEEP     = 2, /* 浅睡眠: 静止60s, 关闭图像传感器 */
    POWER_STATE_DEEP_SLEEP = 3  /* 深度睡眠: 静止300s, 仅BLE广播保持 */
} PowerState;

static PowerState g_power_state = POWER_STATE_ACTIVE;
static uint32_t g_idle_counter_ms = 0;
static uint32_t g_last_activity_ms = 0;

/* 各状态下的采样率 (Hz) */
static const uint16_t g_sample_rates[] = { 200, 50, 0, 0 };

/* 各状态下的BLE连接间隔 (单位: 1.25ms) */
static const uint16_t g_ble_intervals[] = { 8, 16, 80, 400 };

void power_manager_tick(uint32_t current_ms) {
    bool activity = pressure_is_pen_down() || imu_detect_motion();

    if (activity) {
        g_last_activity_ms = current_ms;
        if (g_power_state != POWER_STATE_ACTIVE) {
            power_transition_to(POWER_STATE_ACTIVE);
        }
        return;
    }

    uint32_t idle_ms = current_ms - g_last_activity_ms;

    if (idle_ms > 300000 && g_power_state != POWER_STATE_DEEP_SLEEP) {
        power_transition_to(POWER_STATE_DEEP_SLEEP);
    } else if (idle_ms > 60000 && g_power_state == POWER_STATE_IDLE) {
        power_transition_to(POWER_STATE_SLEEP);
    } else if (idle_ms > 10000 && g_power_state == POWER_STATE_ACTIVE) {
        power_transition_to(POWER_STATE_IDLE);
    }
}

static void power_transition_to(PowerState new_state) {
    PowerState old_state = g_power_state;
    g_power_state = new_state;

    /* 调整采样率 */
    camera_set_sample_rate(g_sample_rates[new_state]);

    /* 调整BLE连接间隔 */
}
```

```

ble_set_connection_interval(g_ble_intervals[new_state]);

/* 状态特定操作 */
switch (new_state) {
    case POWER_STATE_SLEEP:
        camera_power_down();          /* 关闭图像传感器 */
        imu_set_low_power_mode(true); /* IMU进入低功耗模式 */
        break;
    case POWER_STATE_DEEP_SLEEP:
        /* 额外: 关闭LED、降低CPU频率到最低档 */
        led_set_state(LED_STATE_OFF);
        cpu_set_frequency(CPU_FREQ_LOW);
        break;
    case POWER_STATE_ACTIVE:
        if (old_state >= POWER_STATE_SLEEP) {
            camera_power_up();
            imu_set_low_power_mode(false);
        }
        if (old_state == POWER_STATE_DEEP_SLEEP) {
            cpu_set_frequency(CPU_FREQ_HIGH);
        }
        led_set_state(LED_STATE_WRITING);
        break;
    default:
        break;
}

LOG_INFO("Power state: %d -> %d", old_state, new_state);
}

```

E.5 Flash离线缓存 (FIFO环形缓冲区)

```

/* cache/flash_cache.c - SPI Flash环形缓冲区 (WAL模式) */

#define FLASH_SECTOR_SIZE    4096        /* 4KB扇区 */
#define CACHE_SECTORS       128         /* 共128个扇区 = 512KB缓存 */
#define CACHE_HEADER_MAGIC   0xA55A0001 /* 缓存头魔数, 用于完整性检验 */

typedef struct {
    uint32_t magic;          /* 魔数: 0xA55A0001 */
    uint16_t data_len;       /* 数据长度 (字节) */
    uint16_t checksum;       /* 数据CRC16校验 */
    uint32_t timestamp;      /* 数据时间戳 */
    uint8_t  flags;          /* 标志位: Bit0=已确认接收 */
    uint8_t  reserved[3];
} FlashCacheHeader; /* 12字节头部 */

typedef struct {
    uint32_t write_sector; /* 当前写扇区索引 */
    uint32_t read_sector;  /* 当前读扇区索引 */
    uint32_t write_offset; /* 当前写扇区内偏移 */
    uint32_t total_cached; /* 总缓存字节数 */
} FlashCacheState;

```

```

static FlashCacheState g_cache_state;

/**
 * @brief 写入笔迹数据到Flash缓存
 * @param data      笔迹数据指针
 * @param len       数据长度
 * @return 0成功, -ENOMEM缓存满
 */
int flash_cache_write(const uint8_t *data, uint16_t len) {
    if (flash_cache_is_full()) {
        LOG_WARN("Flash cache full, dropping oldest sector");
        /* 覆盖最旧的扇区 (环形FIFO) */
        g_cache_state.read_sector =
            (g_cache_state.read_sector + 1) % CACHE_SECTORS;
    }

    /* 检查当前扇区剩余空间 */
    uint16_t needed = sizeof(FlashCacheHeader) + len;
    uint16_t remaining = FLASH_SECTOR_SIZE - g_cache_state.write_offset;

    if (needed > remaining) {
        /* 移到下一扇区, 擦除后写 */
        g_cache_state.write_sector =
            (g_cache_state.write_sector + 1) % CACHE_SECTORS;
        g_cache_state.write_offset = 0;
        flash_erase_sector(g_cache_state.write_sector * FLASH_SECTOR_SIZE);
    }

    /* 构建缓存头 */
    FlashCacheHeader hdr = {
        .magic      = CACHE_HEADER_MAGIC,
        .data_len   = len,
        .checksum   = crc16(data, len),
        .timestamp  = rtc_get_timestamp(),
        .flags      = 0,
    };

    uint32_t addr = g_cache_state.write_sector * FLASH_SECTOR_SIZE
        + g_cache_state.write_offset;

    /* 写入头部和数据 */
    flash_write(addr, (uint8_t*)&hdr, sizeof(hdr));
    flash_write(addr + sizeof(hdr), data, len);

    g_cache_state.write_offset += needed;
    g_cache_state.total_cached += len;
    return 0;
}

```

附录F 生产测试与质量保证

F.1 固件烧录与出厂自检

智能点阵笔在生产阶段通过SWD（Serial Wire Debug）接口烧录固件，烧录完成后自动执行出厂自检程序：

测试项目	判断标准	失败处理
Flash读写测试	全擦全写无错误	报废
SRAM完整性测试	全0/全1/棋盘格无错误	报废
图像传感器测试	采集图像中点数≥50	更换传感器
压力传感器测试	ADC值在正常范围内	更换传感器
BLE射频测试	RSSI≥-70dBm@1m	天线返修
电池充放电测试	充满容量≥标称95%	更换电池
IMU测试	三轴加速度/陀螺仪数值正常	更换IMU
LED指示灯测试	RGB三色正常点亮	更换LED

F.2 固件版本管理

```
/* version.h - 固件版本定义 */
#define FW_VERSION_MAJOR 1
#define FW_VERSION_MINOR 0
#define FW_VERSION_PATCH 0
#define FW_VERSION_BUILD 20260214

#define FW_VERSION_STRING "1.0.0-20260214"

/* 版本比较宏（用于OTA升级判断） */
#define FW_VERSION_CODE ((FW_VERSION_MAJOR << 24) | \
                        (FW_VERSION_MINOR << 16) | \
                        (FW_VERSION_PATCH << 8))
```

文档编制：深圳自然写科技有限公司 硬件/固件研发部

文档版本：V1.0（附录更新）

最后更新：2026年2月14日

版权所有 © 2026 深圳自然写科技有限公司

附录G RTOS任务设计与调度

G.1 FreeRTOS任务清单

智能点阵笔固件基于FreeRTOS实现多任务并发调度，各任务优先级和栈大小如下：

任务名	优先级	栈大小	说明
ink_capture_task	5（最高）	2048字节	相机采集+点阵解码（200Hz）
pressure_task	4	512字节	压力传感器采样（200Hz）
ble_tx_task	3	1024字节	BLE数据发送队列
ble_rx_task	3	512字节	BLE控制指令接收
flash_cache_task	2	1024字节	Flash离线缓存写入
battery_task	1	256字节	电量监测（1Hz）
power_manage_task	1	256字节	功耗状态机（1Hz）
led_task	0（最低）	256字节	LED状态指示

G.2 任务通信设计

任务间通信使用FreeRTOS消息队列（Queue）和信号量（Semaphore），避免共享内存竞争：

```
/* task_comms.h - 任务间通信接口 */

/* 笔迹数据队列 (ink_capture_task → ble_tx_task / flash_cache_task) */
extern QueueHandle_t g_ink_point_queue; /* 容量: 100个InkPoint */

/* BLE发送完成信号（避免发送缓冲区溢出）*/
extern SemaphoreHandle_t g_ble_tx_ready_sem;

/* 控制指令队列 (ble_rx_task → 各功能任务) */
extern QueueHandle_t g_ctrl_cmd_queue; /* 容量: 10条控制指令 */

/* 功耗状态变更通知 */
extern EventGroupHandle_t g_power_event_group;
#define EVT_ENTER_ACTIVE    (1 << 0)
#define EVT_ENTER_IDLE     (1 << 1)
#define EVT_ENTER_SLEEP    (1 << 2)
```

G.3 中断处理与任务唤醒

```
/* interrupt/camera_irq.c - 相机帧中断处理 */

/* 相机帧就绪中断（VSYNC信号上升沿触发）*/
void CAMERA_VSYNC_IRQHandler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* 通过信号量通知ink_capture_task（ISR安全版本）*/
```

```

xSemaphoreGiveFromISR(g_camera_frame_ready_sem, &xHigherPriorityTaskWoken);

/* 如果唤醒了更高优先级任务，请求任务切换 */
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

/* 笔迹采集任务主循环 */
void ink_capture_task(void *param) {
    InkPoint point;
    DotDecodeResult decode_result;

    for (;;) {
        /* 等待相机帧就绪信号（阻塞，不消耗CPU） */
        xSemaphoreTake(g_camera_frame_ready_sem, portMAX_DELAY);

        /* 读取相机帧 */
        uint8_t *frame = camera_get_latest_frame();

        /* 点阵解码 */
        if (dot_codec_decode(frame, &decode_result) == 0) {
            point.x = (uint16_t)(decode_result.abs_x / PAPER_WIDTH * 65535);
            point.y = (uint16_t)(decode_result.abs_y / PAPER_HEIGHT * 65535);
            point.pressure = pressure_read_normalized();
            point.timestamp = rtc_get_ms();
            point.flags = pressure_is_pen_down() ? 0 : 0x01; /* Bit0=抬笔 */

            /* 发送到笔迹队列（非阻塞，队列满则丢帧） */
            xQueueSendToBack(g_ink_point_queue, &point, 0);
        }
    }
}

```

G.4 LED状态指示定义

LED状态	颜色	闪烁方式	含义
LED_STATE_OFF	熄灭	—	关机或深度睡眠
LED_STATE_BOOT	白色	常亮	启动中
LED_STATE_SCANNING	蓝色	快闪（200ms）	蓝牙扫描广播中
LED_STATE_CONNECTING	蓝色	慢闪（1000ms）	正在连接网关
LED_STATE_CONNECTED	蓝色	常亮	已连接网关
LED_STATE_WRITING	绿色	常亮	书写中（压力传感器触发）
LED_STATE_LOW_BATTERY	红色	慢闪（2000ms）	电量低（<15%）
LED_STATE_CHARGING	橙色	呼吸灯	充电中
LED_STATE_CHARGE_FULL	绿色	常亮	充电完成

LED状态	颜色	闪烁方式	含义
LED_STATE_OTA	紫色	快闪（500ms）	OTA升级中（请勿关机）
LED_STATE_ERROR	红色	3连闪	系统错误

G.5 固件安全措施

安全措施	实现方式	说明
Flash读保护	STM32 RDP Level 1	防止通过调试口读取Flash内容
OTA签名验证	RSA-2048 + SHA-256	只接受官方签名的固件包
通信加密	BLE连接层加密（AES-128 CCM）	防止空中截获笔迹数据
设备绑定	AppKey-DeviceID绑定验证	防止伪造设备接入系统
Bootloader保护	独立分区+写保护	防止OTA意外破坏Bootloader

本文档版权归深圳自然写科技有限公司所有，所有技术细节仅用于软件著作权登记鉴别，请勿用于其他商业用途。

附录F 补充技术规格

F.1 点阵码高速解码优化

F.1.1 SIMD加速解码

```
// dotmatrix_simd.c - ARM NEON加速点阵码解码
#include <arm_neon.h>

// 一次处理16字节像素数据
void decode_row_neon(const uint8_t* pixels, int width,
                    uint8_t* bits_out) {
    const uint8x16_t threshold = vdupq_n_u8(128);
    int x = 0;

    for (; x + 16 <= width; x += 16) {
        uint8x16_t px = vld1q_u8(pixels + x);
        // 与阈值比较：大于128为白(0)，小于等于128为黑(1)
        uint8x16_t result = vcleq_u8(px, threshold);
        // 提取高位构成位掩码
        uint8_t mask = 0;
```



```

        for (int i = 0; i < 16; i++) {
            if (result[i]) mask |= (1 << (i % 8));
            if (i == 7) bits_out[x/8] = mask, mask = 0;
        }
        bits_out[x/8 + 1] = mask;
    }

    // 处理剩余像素
    for (; x < width; x++) {
        if (pixels[x] <= 128) bits_out[x/8] |= (1 << (x % 8));
    }
}

```

F.2 低功耗蓝牙广播优化

F.2.1 自适应广播间隔

```

// ble_adv_manager.c
#define ADV_INTERVAL_FAST_MS    100    // 连接前快速广播
#define ADV_INTERVAL_SLOW_MS    1000   // 长时间未连接慢速广播
#define ADV_FAST_TIMEOUT_S      30     // 30秒后切换到慢速

typedef enum {
    ADV_STATE_OFF = 0,
    ADV_STATE_FAST,
    ADV_STATE_SLOW
} adv_state_t;

static adv_state_t g_adv_state = ADV_STATE_OFF;
static uint32_t g_fast_adv_start_tick = 0;

void ble_adv_update(void) {
    if (g_adv_state == ADV_STATE_OFF) return;

    uint32_t elapsed_s = (HAL_GetTick() - g_fast_adv_start_tick) / 1000;

    if (g_adv_state == ADV_STATE_FAST && elapsed_s >= ADV_FAST_TIMEOUT_S) {
        // 切换到慢速广播节省电量
        ble_gap_adv_stop();

        ble_gap_adv_params_t params = {
            .type = BLE_GAP_ADV_TYPE_CONNECTABLE_UNDIRECTED,
            .interval_min = ADV_INTERVAL_SLOW_MS * 8 / 5, // 单位0.625ms
            .interval_max = ADV_INTERVAL_SLOW_MS * 8 / 5 + 16,
            .channel_mask = 0x07
        };
        ble_gap_adv_start(&params);
        g_adv_state = ADV_STATE_SLOW;
        LOG_INFO("BLE adv switched to slow mode, current=%dmA",
            power_measure_current_ua() / 1000);
    }
}

```

```

void ble_adv_start_fast(void) {
    ble_gap_adv_params_t params = {
        .type = BLE_GAP_ADV_TYPE_CONNECTABLE_UNDIRECTED,
        .interval_min = ADV_INTERVAL_FAST_MS * 8 / 5,
        .interval_max = ADV_INTERVAL_FAST_MS * 8 / 5 + 16,
        .channel_mask = 0x07
    };
    ble_gap_adv_start(&params);
    g_adv_state = ADV_STATE_FAST;
    g_fast_adv_start_tick = HAL_GetTick();
}

```

F.3 电量精确估算算法

F.3.1 库仑计积分法

```

// battery_gauge.c
#define BATTERY_CAPACITY_MAH    200.0f // 电池容量200mAh
#define SAMPLE_INTERVAL_MS     100    // 采样间隔100ms

typedef struct {
    float soc_percent;           // 剩余电量百分比
    float voltage_mv;           // 当前电压 (mV)
    float current_ma;           // 当前电流 (mA, 放电为正)
    float accumulated_mah;      // 已消耗容量 (mAh)
    uint32_t last_sample_tick;  // 上次采样时间
} battery_state_t;

static battery_state_t g_battery;

void battery_gauge_update(void) {
    uint32_t now = HAL_GetTick();
    uint32_t dt_ms = now - g_battery.last_sample_tick;
    if (dt_ms < SAMPLE_INTERVAL_MS) return;

    // 采样ADC
    g_battery.voltage_mv = adc_read_voltage();
    g_battery.current_ma = adc_read_current();

    // 积分: 累计消耗容量 = 电流(mA) × 时间(h)
    float dt_h = dt_ms / 3600000.0f;
    g_battery.accumulated_mah += g_battery.current_ma * dt_h;

    // SOC = 1 - 已消耗/总容量
    g_battery.soc_percent =
        (1.0f - g_battery.accumulated_mah / BATTERY_CAPACITY_MAH) * 100.0f;
    g_battery.soc_percent = CLAMP(g_battery.soc_percent, 0.0f, 100.0f);

    // 电压修正 (防止长期误差累积)
    float ocv_soc = voltage_to_soc_ocv(g_battery.voltage_mv);
    if (fabsf(ocv_soc - g_battery.soc_percent) > 10.0f) {
        // 差异超过10%时用OCV校正
        g_battery.soc_percent = 0.8f * g_battery.soc_percent + 0.2f * ocv_soc;
    }
}

```

```

    }

    g_battery.last_sample_tick = now;
}

// OCV (开路电压) 与SOC对应表
static const float OCV_TABLE[][2] = {
    {3200, 0}, {3400, 5}, {3500, 10}, {3600, 20},
    {3650, 30}, {3700, 50}, {3750, 70}, {3800, 85},
    {3850, 95}, {3900, 100}
};

float voltage_to_soc_ocv(float voltage_mv) {
    int n = sizeof(OCV_TABLE) / sizeof(OCV_TABLE[0]);
    if (voltage_mv <= OCV_TABLE[0][0]) return 0.0f;
    if (voltage_mv >= OCV_TABLE[n-1][0]) return 100.0f;

    for (int i = 1; i < n; i++) {
        if (voltage_mv <= OCV_TABLE[i][0]) {
            float t = (voltage_mv - OCV_TABLE[i-1][0]) /
                (OCV_TABLE[i][0] - OCV_TABLE[i-1][0]);
            return OCV_TABLE[i-1][1] + t * (OCV_TABLE[i][1] - OCV_TABLE[i-1][1]);
        }
    }
    return 100.0f;
}

```

附录G 补充技术规格

G.1 RTOS任务优先级配置

```

// rtos_config.c - FreeRTOS任务优先级与堆栈配置
#define TASK_PRIO_SENSOR_READ    7    // 最高: 传感器数据读取
#define TASK_PRIO_INK_ENCODE     6    // 高: 笔迹编码
#define TASK_PRIO_BLE_TX        5    // 高: BLE数据发送
#define TASK_PRIO_POWER_MGMT     4    // 中: 电源管理
#define TASK_PRIO_CACHE_FLUSH    3    // 中低: 缓存刷新
#define TASK_PRIO_STATUS_LED     2    // 低: 状态LED控制
#define TASK_PRIO_IDLE          1    // 最低: 空闲任务

// 任务堆栈大小 (单位: 字节)
#define STACK_SENSOR_READ        512
#define STACK_INK_ENCODE         1024
#define STACK_BLE_TX             768
#define STACK_POWER_MGMT         512
#define STACK_CACHE_FLUSH        512

void create_all_tasks(void) {
    xTaskCreate(sensor_read_task, "SensorRead",
        STACK_SENSOR_READ / sizeof(StackType_t),
        NULL, TASK_PRIO_SENSOR_READ, &g_sensor_task_handle);
}

```

```

xTaskCreate(ink_encode_task, "InkEncode",
            STACK_INK_ENCODE / sizeof(StackType_t),
            NULL, TASK_PRI0_INK_ENCODE, &g_encode_task_handle);

xTaskCreate(ble_tx_task, "BleTx",
            STACK_BLE_TX / sizeof(StackType_t),
            NULL, TASK_PRI0_BLE_TX, &g_ble_tx_task_handle);

xTaskCreate(power_mgmt_task, "PowerMgmt",
            STACK_POWER_MGMT / sizeof(StackType_t),
            NULL, TASK_PRI0_POWER_MGMT, &g_power_task_handle);

xTaskCreate(cache_flush_task, "CacheFlush",
            STACK_CACHE_FLUSH / sizeof(StackType_t),
            NULL, TASK_PRI0_CACHE_FLUSH, &g_cache_task_handle);
}

```

G.2 笔压标定算法

```

// pressure_calibration.c
#define CALIB_POINTS    5    // 标定点数量
#define CALIB_ADC_BITS 12    // ADC位数 (0-4095)

typedef struct {
    uint16_t adc_raw[CALIB_POINTS];    // ADC原始值
    float    force_gram[CALIB_POINTS]; // 对应压力 (克)
    float    slope;                    // 线性拟合斜率
    float    intercept;                // 线性拟合截距
    bool     is_calibrated;
} pressure_calib_t;

static pressure_calib_t g_calib;

// 最小二乘法线性拟合
void pressure_calibration_fit(void) {
    float sum_x = 0, sum_y = 0, sum_xy = 0, sum_x2 = 0;
    int n = CALIB_POINTS;

    for (int i = 0; i < n; i++) {
        float x = g_calib.adc_raw[i];
        float y = g_calib.force_gram[i];
        sum_x += x;
        sum_y += y;
        sum_xy += x * y;
        sum_x2 += x * x;
    }

    float denom = n * sum_x2 - sum_x * sum_x;
    if (fabsf(denom) < 1e-6f) {
        LOG_ERROR("标定失败: ADC值无变化");
        return;
    }
}

```

```

    g_calib.slope      = (n * sum_xy - sum_x * sum_y) / denom;
    g_calib.intercept = (sum_y - g_calib.slope * sum_x) / n;
    g_calib.is_calibrated = true;

    LOG_INF0("压力标定完成: slope=%.4f, intercept=%.2f",
            g_calib.slope, g_calib.intercept);
}

float pressure_adc_to_gram(uint16_t adc_raw) {
    if (!g_calib.is_calibrated) return adc_raw / 4095.0f * 500.0f; // 默认映射
    float gram = g_calib.slope * adc_raw + g_calib.intercept;
    return CLAMP(gram, 0.0f, 600.0f);
}

uint8_t pressure_gram_to_normalized(float gram) {
    // 映射到0-255, 最大压力600克
    return (uint8_t)CLAMP(gram / 600.0f * 255.0f, 0.0f, 255.0f);
}

```

附录H 补充技术规格

H.1 倾斜角计算

```

// tilt_calculator.c
// 利用IMU（三轴加速度计）计算笔的倾斜角
#include <math.h>

typedef struct {
    float x, y, z; // 加速度计原始值（单位：g）
} accel_t;

typedef struct {
    float elevation; // 仰角（0°=水平，90°=垂直）
    float azimuth;   // 方位角（0°=正前方）
} pen_tilt_t;

pen_tilt_t calculate_tilt(const accel_t* accel) {
    pen_tilt_t result;

    // 计算仰角: arctan(z / sqrt(x^2 + y^2))
    float xy_magnitude = sqrtf(accel->x * accel->x + accel->y * accel->y);
    result.elevation = atan2f(accel->z, xy_magnitude) * 180.0f / M_PI;

    // 计算方位角: arctan2(y, x)
    result.azimuth = atan2f(accel->y, accel->x) * 180.0f / M_PI;
    if (result.azimuth < 0) result.azimuth += 360.0f;

    return result;
}

// 滑动平均滤波消抖（窗口大小=8）

```

```

#define TILT_FILTER_SIZE 8
static pen_tilt_t tilt_history[TILT_FILTER_SIZE];
static int tilt_idx = 0;

pen_tilt_t tilt_filtered(pen_tilt_t raw) {
    tilt_history[tilt_idx % TILT_FILTER_SIZE] = raw;
    tilt_idx++;

    pen_tilt_t sum = {0};
    int count = tilt_idx < TILT_FILTER_SIZE ? tilt_idx : TILT_FILTER_SIZE;
    for (int i = 0; i < count; i++) {
        sum.elevation += tilt_history[i].elevation;
        sum.azimuth += tilt_history[i].azimuth;
    }
    return (pen_tilt_t){ sum.elevation / count, sum.azimuth / count };
}

```

H.2 NFC标签读取

```

// nfc_reader.c - 读取点阵纸NFC标签获取页面ID
#include "nfc_hal.h"

#define NFC_PAGE_ID_BLOCK 4 // 页面ID存储在NDEF块4

bool nfc_read_page_id(uint32_t* page_id_out) {
    nfc_tag_t tag;

    // 检测NFC标签
    if (!nfc_hal_detect(&tag, 100 /* timeout_ms */)) {
        return false;
    }

    // 验证标签类型 (MIFARE Ultralight)
    if (tag.type != NFC_TAG_MIFARE_UL) {
        LOG_WARN("不支持的NFC标签类型: %d", tag.type);
        return false;
    }

    // 读取页面ID块 (4字节)
    uint8_t data[4];
    if (!nfc_hal_read_block(&tag, NFC_PAGE_ID_BLOCK, data)) {
        LOG_ERROR("NFC读取失败");
        return false;
    }

    // 大端序解析
    *page_id_out = ((uint32_t)data[0] << 24) |
        ((uint32_t)data[1] << 16) |
        ((uint32_t)data[2] << 8) |
        ((uint32_t)data[3]);

    LOG_DEBUG("NFC读取页面ID: 0x%08X", *page_id_out);
}

```

```
    return true;  
}
```

本文档版权归深圳自然写科技有限公司所有，所有技术细节仅用于软件著作权登记鉴别，请勿用于其他商业用途。