# 果蝇1号 DF-01 开发者手册与使用教程

**果蝇一号（DF-01）是一体化 13.56 MHz 高性能非接触式 RFID / NFC 读卡模块**：板载微带天线、高灵敏度射频前端与智能控制核心全部集成在单块紧凑型 PCB 上，对外仅暴露 5 个 2.54 mm 标准引脚。上电即自动开启射频感应场独立工作，无需任何复杂初始化配置；同时支持通过一路标准 TTL 异步串口与主控板或 PC 通信，支持读取卡号、读写数据块、自主免轮询上报、硬件防重读过滤以及外部蜂鸣器/指示灯联动。

| 文档信息 | 内容说明 |
| --- | --- |
| **文档版本** | v2.1 · 2026-10 全新教程与实战修订版 |
| **适用产品** | 果蝇一号 DF-01（Drosophila-01 一体化 RFID/NFC 读卡模块） |
| **射频频段** | 13.56 MHz 高频（支持 ISO/IEC 14443 Type A、MIFARE Classic 1K / 4K、NTAG 系列） |
| **通信接口** | 3.3V TTL 异步串口（默认 115200 8N1，支持 2400~115200 可调） |
| **协议基线** | 标准 `7F` 串口通信协议基线 · 参数持久化 v15 |
| **配套工具** | DF-01 Studio 跨平台可视化工作台 / 通用串口调试助手 |
| **适用读者** | 创客、嵌入式工程师、软硬件开发者、系统集成商、技术爱好者 |
| **内容范围** | 开箱上手 → 原理与硬件规格 → 7F 协议详解 → 配套工作台 → 实战开发 → 完整指令集 → 验收与排障 |

**说明与背景**

**阅读约定与快速指引**

- 报文字节、长度 `LEN`、命令字 `CMD`、状态字 `STATUS` 与数据参数一律以**十六进制**书写（如 `7F`、`10`、`90`、`00`）。
- 时间（ms、s）、通信波特率（115200 bps）、字节个数均为**十进制**。
- 参数偏移量（Offset）从 **0** 开始计数；多字节整型字段默认为**小端序**（Little-Endian，低位字节在前，高位字节在后）。
- 文中报文示例均以模块出厂默认地址 `ID = 00` 给出；若已修改地址，请把帧中的 `ID` 字节替换为新地址。

# 目录


- 第 0 章：五分钟极速开箱上手（初学者极速指南）
- 0.1 认识模块与外观全貌
- 0.2 极简接线：只需 4 根线
- 0.3 上电指示灯状态自检与硬件联动作时序
- 0.4 第一次成功刷卡：串口助手 10 秒看懂输出
- 0.5 为什么选择一体化读卡模块?
- 第 1 章：系统工作原理与硬件规格
- 1.1 13.56 MHz RFID/NFC 感应原理通俗讲
- 1.2 电气规格与性能指标全表
- 1.3 板载指示灯与外部硬件联动驱动
- 1.4 语音播报输出能力（U13T 协议兼容）
- 第 2 章：7F 串口通信协议详解（通俗教程）
- 2.1 为什么设计 7F 协议?
- 2.2 协议帧解剖与格式规范
- 2.3 核心亮点：0x7F 自动双写转义机制
- 2.4 通俗异或校验和（XOR）怎么算?
- 2.5 状态码 STATUS 一览表
- 2.6 全自动工作模式（Auto Mode）与业务分发
- 2.7 硬件级防重读机制（Anti-Reread）
- 2.8 射频增益（RF Gain）调谐技巧
- 2.9 一键读取模块全参数（B1 帧结构全面拆解）
- 第 3 章：配套工作台 DF-01 Studio 深度使用教程
- 3.1 跨平台工作台概览（免安装，即开即用）
- 3.2 模拟模式与串口实物连接
- 3.3 读卡工作台核心功能
- 3.4 多编码文本与数据解析工具
- 3.5 密钥管理与卡片格式化
- 3.6 四套专业设计主题切换
- 第 4 章：实战应用与代码复用（从零开发你的读卡项目）
- 4.0 示例源码与压缩包下载
- 4.1 通用 Python 驱动模块（可作为库直接导入）
- 4.2 C / C++ 驱动库（包含头文件并链接实现）
- 4.3 Rust 驱动库（no_std 兼容，可用于嵌入式与 PC）
- 4.4 四大经典工程落地配方
- 第 5 章：完整指令参考手册（开发者权威字典）
- 5.1 指令一览总表
- 5.2 `10` / `90` 单次读取卡号 UID
- 5.3 `11` / `91` 单次读取数据块

- 5.4 `12` / `92` 写入数据块
- 5.5 `2B` / `AB` 装载扇区认证密钥 (Key A / Key B)
- 5.6 `2C` / `AC` 修改工作波特率
- 5.7 `2D` / `AD` 修改模块设备地址 (ID)
- 5.8 `2E` / `AE` 配置自动工作模式
- 5.9 `2F` / `AF` 配置防重读时间
- 5.10 `30` / `B0` 配置射频接收增益档位
- 5.11 `31` / `B1` 读取模块全部当前配置参数
- 5.12 `32` / `B2` 配置扩展输出缓变时间与上电延时
- 5.13 `33` / `B3` 配置扩展输出初始电平/占空比
- 5.14 `34` / `B4` 配置扩展输出启动极性/通道模式
- 5.15 预留指令说明
- 第 6 章：工程化、验收与排障
- 6.1 进料与出厂检验（IQC / OQC 规范测试流程）
- 6.2 常见问题排障指南（FAQ 十问十答）
- 6.3 硬件安装与外壳结构建议
- 附录
- 附录 A：标准请求与应答报文速查表
- 附录 B：MIFARE Classic 1K 存储结构全解
- 附录 C：RFID / NFC 专业术语对照表
- 附录 D：模块参数持久化与出厂配置映射


CHAPTER

# 第 0 章：五分钟极速开箱上手（初学者极速指南）

## 0.1 认识模块与外观全貌

果蝇一号（DF-01）的设计理念是**"化繁为简、极致集成"**。传统读卡方案通常需要天线板、高频匹配电路板与主控板等多块电路板通过同轴电缆或飞线连接，既笨重又容易引入外界电磁干扰。果蝇一号将 13.56 MHz 环形高频天线、高精度滤波电容、专用射频前端、智能控制芯片、状态指示灯以及外部驱动电路全部熔铸于一块轻巧的黑色 PCB 上。

像它的名字"果蝇"一样，该模块体积小巧、感应极其灵敏（毫秒级瞬时寻卡），模块一侧设计了标准的 5 个 2.54 mm 间距直插排针，能够非常方便地插入面包板、PCB 底板或通过杜邦线与主控直连。




图 0-1　果蝇一号 DF-01 模块实物全貌（背面丝印、接线排针与天线焊盘）

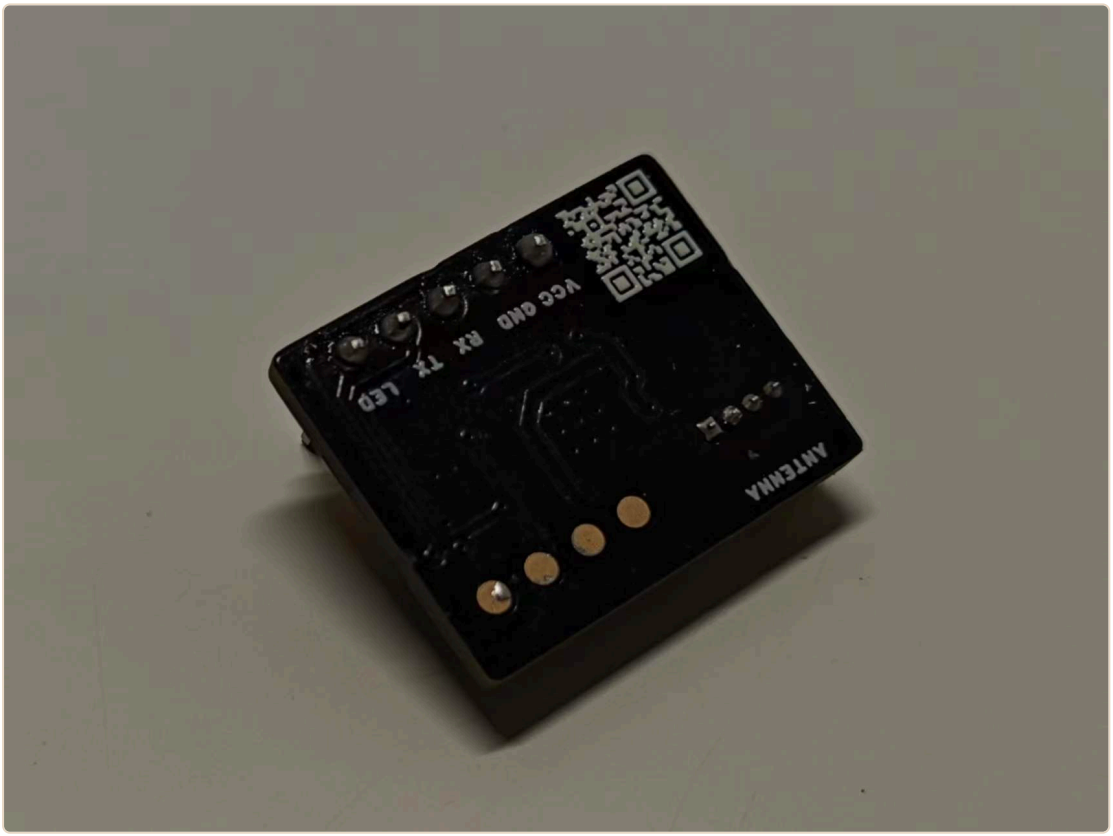



图 0-2　果蝇一号 DF-01 模块实物斜视外观图

在正面，天线环绕整板外周，正中为感应核心区域；在底面，清晰标注了出厂二维码、外部天线扩展焊盘（ANTENNA）以及底部 5 个引脚的名称丝印：`VCC`、`GND`、`RX`、`TX`、`LED`。

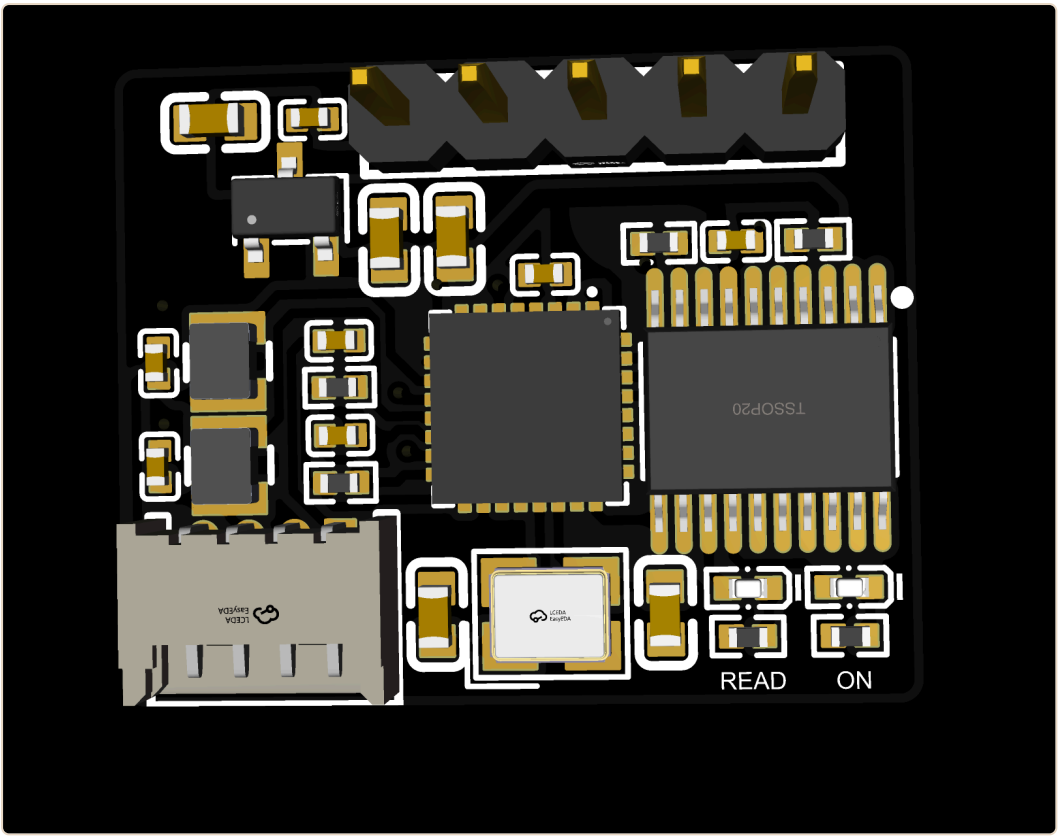



图 0-3　果蝇一号 DF-01 3D PCB 正面元器件与指示灯布局

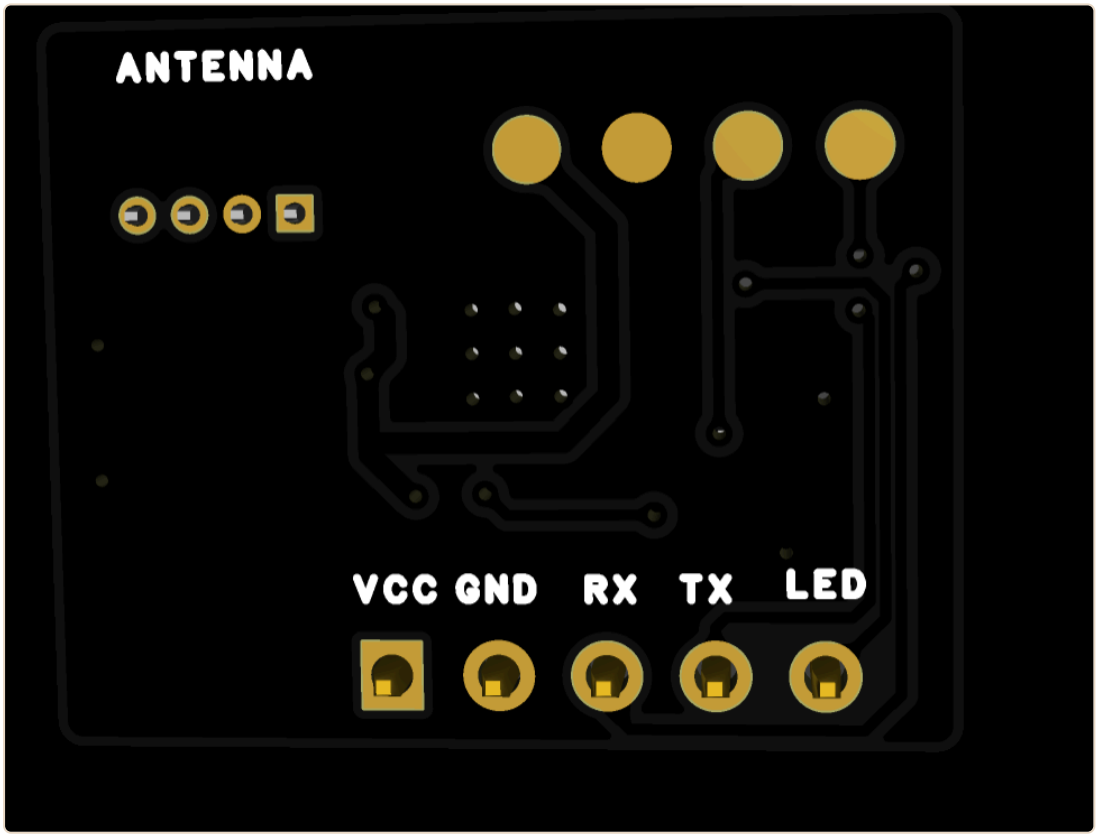



图 0-4　果蝇一号 DF-01 3D PCB 背面引脚与丝印定义

## 0.2 极简接线：只需 4 根线

要让模块工作起来，通常**只需要连接 4 根线**（电源正极、地线、串口收、串口发）。第 5 根 `LED` 引脚为可选的联动触发脚，用于无需主控介入的情况下直接联动外部蜂鸣器、大功率指示灯或电磁锁具。

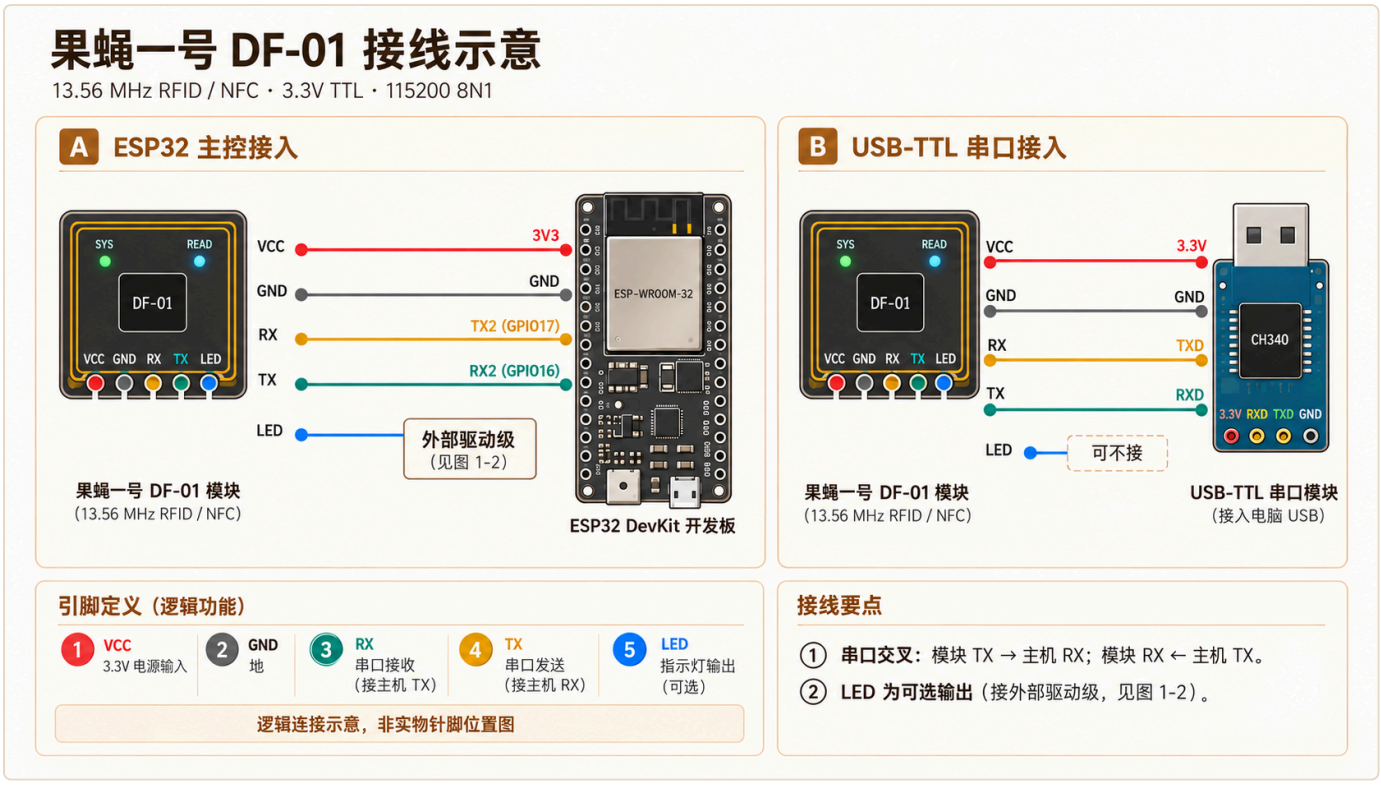



图 0-5 DF-01 仿真接线全景示意图（嵌入式开发板与 USB-TTL 两种场景）

| 引脚编号 | 引脚丝印 | 信号类型 | 电平规格 | 功能说明 | 典型接线目标 |
| --- | --- | --- | --- | --- | --- |
| **Pin 1** | **VCC** | 电源输入 | DC 3.0V ~ 5.5V | 模块主供电输入。板载高精度稳压与去耦电路，工作电流约 35~55 mA | 接 USB-TTL 的 3.3V 或 5V，或主板 3.3V/5V 电源 |
| **Pin 2** | **GND** | 信号参考地 | 0V | 系统接地端，必须与主控系统单点可靠共地 | 接 USB-TTL 或主板的 GND |
| **Pin 3** | **RX** | 串口接收 | 3.3V TTL | 模块数据接收端。接主机的发送引脚（TX） | 接 USB-TTL 的 **TXD**，或主控板的 **UART_TX** |
| **Pin 4** | **TX** | 串口发送 | 3.3V 推挽 | 模块数据发送端。接主机的接收引脚（RX）。推挽输出，不可多机直接硬并联 | 接 USB-TTL 的 **RXD**，或主控板的 **UART_RX** |
| **Pin 5** | **LED** | 联动输出 | 3.3V 推挽 | 硬件联动触发脚。平时为低电平（0V）；刷卡成功时输出 500 ms 的 3.3V 高电平脉冲 | 悬空不接，或接三极管基极驱动蜂鸣器、指示灯、继电器 |

**实用技巧**

**接线口诀：交叉互联，电源莫反**

1. **串口交叉**：模块的 `TX` 必须接主机的 `RX`；模块的 `RX` 必须接主机的 `TX`。最常见的错误就是 TX 接 TX、RX 接 RX，导致串口无法通信。
2. **电平兼容**：模块串口信号为标准 3.3V TTL 逻辑电平。5V 主控直接接收模块 TX 毫无压力；主控 5V TX 发往模块 RX 时，因模块输入端口具备过压钳位能力亦可正常工作，但推荐在 5V 主控 TX 线上串联一个 1kΩ 电阻或电平转换模块以获得最佳长期电气寿命。

## 0.3 上电指示灯状态自检与硬件联动作时序

果蝇一号板载高灵敏度指示灯与硬件引脚协同工作，无论是通过直观声光感知还是通过逻辑分析仪捕捉逻辑波形，都能获得极高确定性的时序流转：

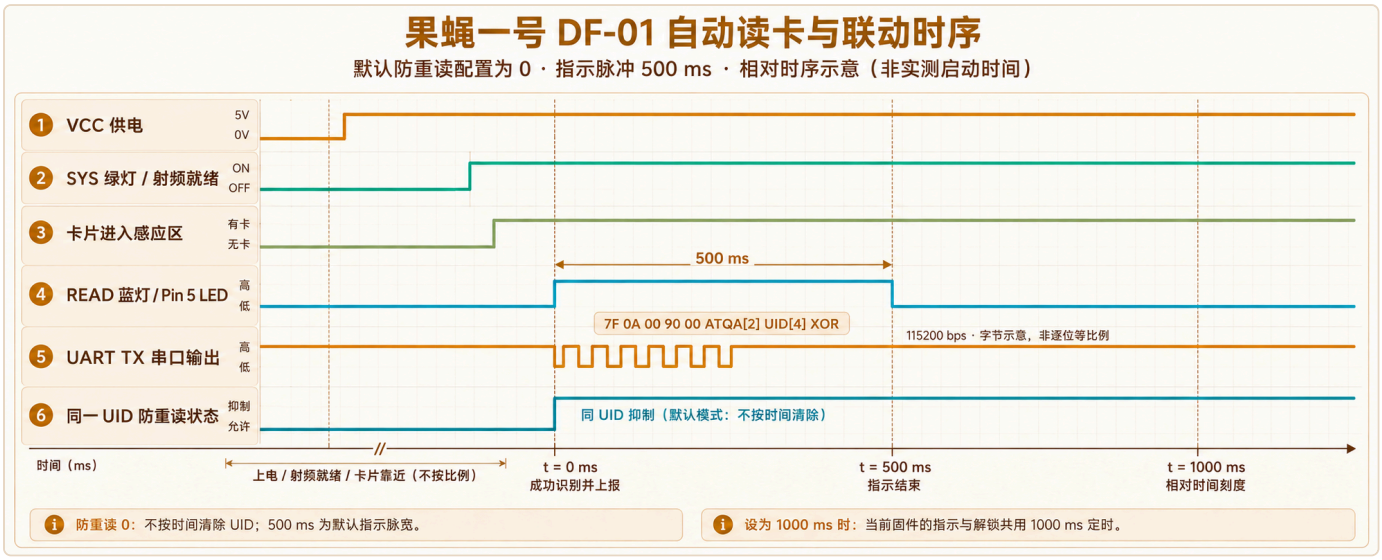



*图 0-6：默认防重读配置为 0 时的自动读卡时序示意；以成功识别为相对时间零点，显示 500 ms 指示脉冲与 115200 bps 串口上报。初始化区间及 UART 字节波形不按实测时间等比例绘制。*

### 果蝇一号 DF-01 完整时序流转与信号动作对照表

| 时序阶段 / 信号轨道 | 信号状态与电平逻辑 | 典型持续时间 | 硬件动作与业务行为说明 |
| --- | --- | --- | --- |
| **T0: 供电启动** | VCC 上升并稳定，系统执行初始化 | 取决于初始化流程 | 内部微处理器完成复位自检，射频前端完成基础寄存器初始化配置；图中不将启动时间固定为 20 ms |
| **T1: 射频待机监听** | SYS 绿灯常亮，13.56MHz 连续发射 | 持续保持 | 板载环形微带天线向空间辐射连续能量载波，系统处于低功耗空闲监听等待状态 |
| **T2: 卡片进入感应场** | 空间磁场耦合，卡片能量激活 | 瞬态（< 15 ms） | 模块与无源卡片完成 ISO14443A 防冲撞握手，成功解析并锁存 4/7 字节物理卡号 UID |
| **T3: 极速串口上报** | UART TX 异步输出 7F 协议帧 | 随转义后的帧长变化 | 自动读 UID 模式打包上报 `7F 0A 00 90 00 [ATQA 2B] [UID 4B] XOR`；图中 UART 波形仅作字节发送示意 |
| **T4: 硬件联动作响应** | READ 蓝灯点亮，Pin 5 输出高电平 | 默认防重读为 0 时保持 **500 ms** | 当前固件共用指示与 UID 解锁定时；例如防重读设为 1000 ms 时，指示时长也为 1000 ms；大负载经图 1-2 的驱动级接入 |
| **T5: 硬件防重读状态** | 记录已自动上报的 UID，抑制相同 UID 重复上报 | 默认 0 不按时间清除；非零值按配置定时 | 图中展示默认配置，不将“500 ms 指示 + 1000 ms 解锁”画成同一配置下的固定行为 |
| **T6: 卡片离开感应区** | 感应区恢复无卡状态 | 取决于卡片移动 | 当前固件不能仅以卡片移开判断 UID 缓存已清除；需要重新读取时，按协议重置状态或使用主动读取流程 |

- **SYS 绿灯（电源与射频指示）**：接入 VCC 与 GND 后，SYS 绿灯常亮代表内部硬件自检正常、射频场已成功建立并开启 13.56 MHz 高频电磁场发射。
- **READ 蓝灯（寻卡状态指示）**：在待机状态下平时处于熄灭状态。当检测到合法 IC 卡靠近感应区并成功读取卡号时，READ 蓝灯会瞬间点亮 **500 毫秒** 后自动熄灭；若卡片一直放置在天线上不拿开，防重读抑制机制启动，指示灯不会持续闪烁。

## 0.4 第一次成功刷卡：串口助手 10 秒看懂输出

如果你手上有一个普通的 USB 转 TTL 串口小板，无需安装任何专用软件，打开电脑上任意的"串口调试助手"即可在 10 秒内体验刷卡：

1. **连接硬件**：将模块的 VCC、GND、RX、TX 分别与 USB-TTL 的 3.3V（或 5V）、GND、TXD、RXD 连接（注意交叉互联：模块 RX 接 TTL 的 TXD，模块 TX 接 TTL 的 RXD）。
2. **打开串口**：插入电脑 USB 接口，在串口助手中选择对应的 COM 端口号，设置参数为：**波特率 115200、数据位 8、停止位 1、校验位 None (8N1)**，勾选"十六进制显示（HEX）"。
3. **贴近卡片**：拿一张常见的 13.56 MHz IC 卡（门禁卡、饭卡、水卡或支持 NFC 模拟的手机），贴近模块正面天线。
4. **见证输出**：板载 READ 灯点亮一次，串口助手接收区瞬间弹出一串十六进制数据。

**报文 / 文本**

```
7F 0A 00 90 00 04 00 E0 45 AF AB 3F
```

这串数据是什么意思？ - `7F`：帧头起始标志。 - `0A`：该帧有效数据长度为 10（0x0A）字节。 - `00`：模块地址 00。 - `90`：读取卡号的成功响应码。 - `00`：操作状态码（00 表示成功）。 - `04 00`：卡片类型标识符（ATQA，标明这是一张标准 MIFARE Classic 卡）。 - `E0 45 AF AB`：卡片的唯一物理卡号（UID），在门禁系统中这就是你的"身份证号"！ - `3F`：整包数据的全段异或校验和（XOR Checksum）。

**说明与背景**

**为什么主机没有发命令，模块就自动返回了数据？** 这正是模块最具人性化的**"全自动工作模式（Auto Mode）"**！传统读卡器需要主机以每秒几十次的频率不断轮询发问"有卡吗？有卡吗？"，严重浪费主控算力和总线带宽。而模块内部集成了全自动寻卡引擎，平时静默待命，卡来即读、读完即报，主机只需在串口接收中断中被动等待数据推送即可。

---

## 0.5 为什么选择一体化读卡模块？

在嵌入式与物联网项目中，很多开发者曾尝试使用廉价的裸射频芯片自制读卡器，但往往深陷泥潭：

| 对比维度 | 传统射频芯片裸片方案（痛点） | 果蝇一号 DF-01 智能一体化方案（优势） |
| --- | --- | --- |
| **天线设计** | ❌ 必须外接天线，阻抗匹配调试极其繁琐 | ✅ **板载精密微带天线**，出厂高 Q 值标定，免调谐 |
| **驱动开发** | ❌ 主控需要移植复杂的底层驱动 | ✅ **极简 7F 串口协议**，几行代码即可读取卡号 |
| **引脚占用** | ❌ 占用主控 6~8 个 SPI/I2C 高频引脚 | ✅ **仅需 2 根串口线（TX/RX）**，主控零负担 |
| **轮询开销** | ❌ 主控必须不间断主动轮询，耗费大量 CPU | ✅ **内部固件自动轮询**，刷卡主动上报 |
| **防重读机制** | ❌ 无法防止重读，刷卡极易发生数据风暴 | ✅ **硬件级防重读冷却**（出厂预设单次锁定） |
| **外设联动** | ❌ 外接蜂鸣器需主控编写额外延时定时器 | ✅ **板载 500ms 硬件联动脉冲**，直驱蜂鸣器与指示灯 |

CHAPTER

# 第 1 章：系统工作原理与硬件规格

## 1.1 13.56 MHz RFID/NFC 感应原理通俗讲

果蝇一号工作在国际通用的 **13.56 MHz ISM 频段**，属于近场高频（HF）非接触式射频识别技术。

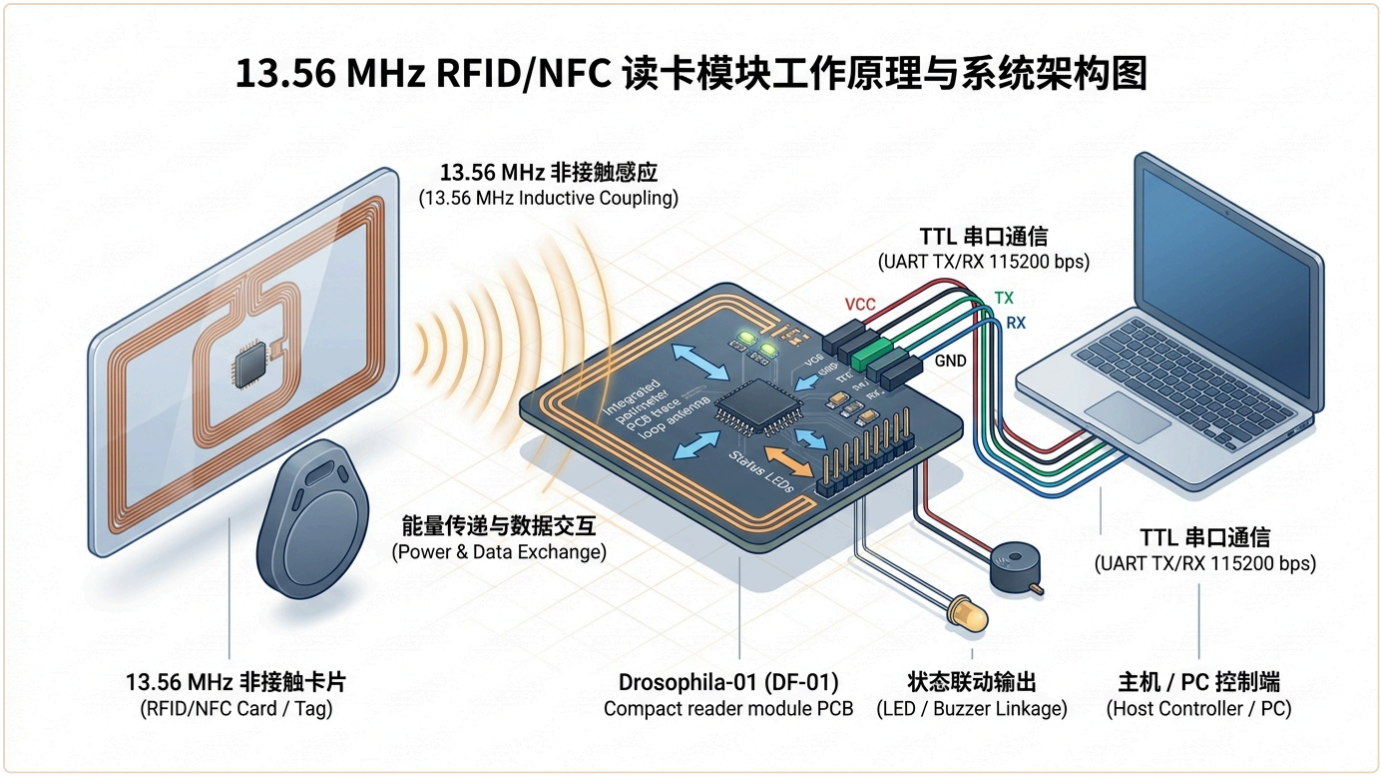



图 1-1　13.56 MHz RFID/NFC 读卡模块工作原理与系统架构图

其能量传递与数据交互的本质是一个**空间松耦合的高频变压器**： 1. **载波建立与磁场发射**：果蝇一号通过板载环形微带天线向空间辐射出交变磁场。 2. **能量捕获（无源供电）**：常见的 IC 卡内部没有任何电池。卡片内部封装有一圈精细的铜线圈天线和一个微型硅芯片。当卡片进入模块辐射的磁场区时，线圈产生感应电动势，经芯片内部整流桥整流并由储能电容平滑为直流电，瞬间"唤醒"了卡片芯片。 3. **载波调制与双向交互**： - **下行数据（读卡器到卡片）**：模块通过对 13.56 MHz 载波进行 100% 或微幅 ASK（幅移键控）振幅调制，将读卡指令发送给卡片。 - **上行数据（卡片到读卡器）**：卡片芯片内部开关控制自身线圈的负载阻抗（负载调制），通过天线互感反作用于读卡器端回路，读卡器解调出这个微小的幅度变化，完成数据读取。 4. **智能处理与串口推送**：模块内部的射频前端捕获到微弱信号后，完成放大、滤波与数字解调，控制核心依次执行"防冲突（Anti-Collision）"、"卡号选择（Select）"与"扇区密钥认证（Authentication）"，最终将完整的卡号或块数据按照 7F 协议封装为串口报文推送给主机。

## 1.2 电气规格与性能指标全表

| 参数分类 | 参数项 | 标称数值 / 规格 | 测试条件 / 补充说明 |
|---|---|---|---|
| **供电特性** | 供电输入电压 | **DC 3.0 V ~ 5.5 V** | 推荐使用 3.3V 或 5.0V 纯净稳压电源 |
| | 待机静态电流 | 约 35 mA | 天线载波发射，等待卡片进入感应场 |
| | 刷卡瞬间峰值电流 | 约 55 mA | 读写数据及指示灯点亮瞬间 |
| **射频指标** | 工作中心频率 | **13.56 MHz** | 符合 ISO/IEC 14443 Type A 国际标准 |
| | 兼容卡片类型 | MIFARE Classic 1K / 4K、Ultralight、NTAG213/215/216 | 支持常见门禁卡、滴胶异形扣、手环及 NFC 手机模拟 UID |
| | 天线感应距离 | **30 mm ~ 55 mm**（标准 ISO 厚卡）<br>**20 mm ~ 35 mm**（微型滴胶钥匙扣） | 射频增益设为出厂最大档（48 dB），非金属环境下测试 |
| | 射频增益调节 | **18 dB ~ 48 dB（共 8 档软件可调）** | 可通过串口指令灵活适应不同安装环境 |
| **串口通信** | 接口逻辑电平 | **3.3 V TTL**（输入耐压 5.5 V） | 兼容 3.3V 与 5.0V 主控系统 |
| | 出厂默认波特率 | **115200 bps** | 8 位数据位，1 位停止位，无奇偶校验 (8N1) |
| | 支持波特率范围 | 2400 ~ 115200 bps | 支持 2400、4800、9600、14400、19200、28800、38400、57600、115200 bps |
| **联动接口** | LED 联动引脚 | **3.3 V 推挽输出**（最大灌拉电流 12 mA） | 刷卡成功输出 500 ms 脉冲，可直驱光耦或驱动三极管 |
| **物理规格** | 模块几何外形 | 约 35 mm × 35 mm 正方形超薄板 | 板载天线与控制电路一体化成型 |

| 参数分类 | 参数项 | 标称数值 / 规格 | 测试条件 / 补充说明 |
| --- | --- | --- | --- |
| | 排针间距 | 2.54 mm 标准直插间距（5 Pin） | 方便插入标准面包板与杜邦线接插 |
| | 工作环境温度 | -20 ℃ ~ +70 ℃ | 工业级耐受范围 |

## 1.3 板载指示灯与外部硬件联动驱动

果蝇一号模块设计了极其实用的外部联动能力：很多应用场景（例如门禁刷卡、储物柜开锁、售货机刷卡）都需要在刷卡成功后立刻"哔"一声提示音或者"嗒"一声开启门锁继电器。如果所有驱动都要由主控在解析串口后通过 IO 口完成，不仅代码冗长，而且存在串口传输和处理延迟。

为此，果蝇一号将第 5 引脚定义为独立的 **LED 联动输出端**：

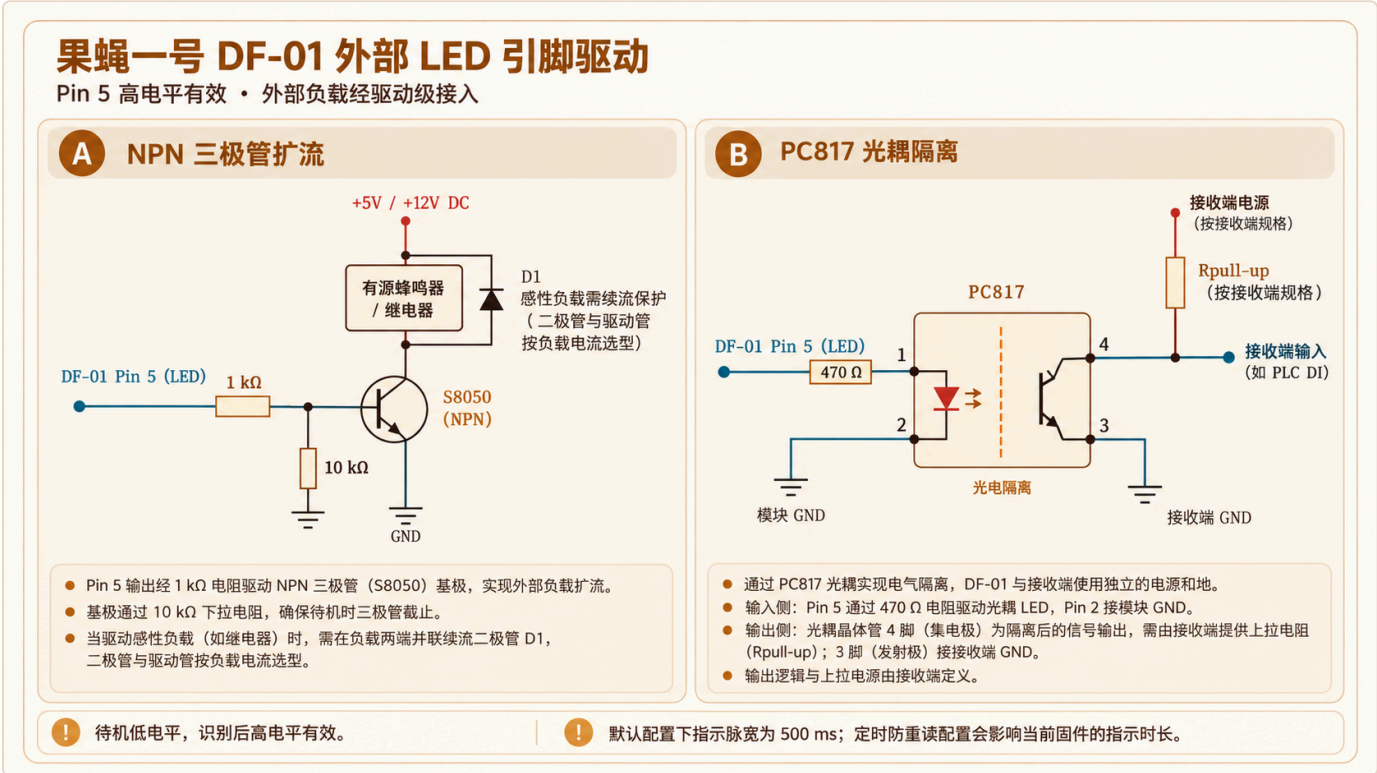



*图 1-2：DF-01 外部引脚驱动设计（NPN 扩流与 PC817 光耦隔离示例；负载、电阻及续流保护器件按实际接收端与负载规格选型）*

- **工作机制**：在待机状态下，LED 引脚保持 0V 低电平；当模块检测到合法卡片并成功读出卡号或数据块时，内部硬件自动将 LED 引脚拉高为 3.3V，持续点亮 **500 毫秒** 后自动恢复为低电平。
- **直驱能力**：该引脚内部为强推挽输出结构，高电平驱动能力为 12 mA，可直接驱动普通发光二极管（串联 1kΩ 限流电阻）、光电耦合器或主控外部中断引脚。
- **大负载驱动**：若需驱动 5V/12V 有源蜂鸣器或电磁铁，串联一个 1kΩ 基极电阻并外接一颗通用 NPN 三极管，即可实现零延迟刷卡即响、刷卡即开门效果。

## 1.4 语音播报输出能力（U13T 协议兼容）

除了串口报文与电平联动外，模块还内置了业界广泛使用的 **U13T 离线语音播报协议引擎**（出厂模式 `03`）： - 当开启语音模式后，模块读取扇区数据块后，会自动把块内的文字内容格式化为 U13T 专用的 `0xFD` 语音控制帧。 - 如果将模块的 TX 引脚直接连接到市面上常见的离线语音播报模块的 RX 引脚，无需主控编写任何合成指令，刷卡即可直接听到清脆的真人语音发音（如"欢迎光临"、"刷卡成功"）。

CHAPTER

# 第 2 章：7F 串口通信协议详解（通俗教程）

## 2.1 为什么设计 7F 协议?

在嵌入式通信中，最忌讳的是"裸奔的原始数据"：如果串口只发送原始的 4 字节卡号，一旦通信线受到电磁干扰产生一个毛刺字节，接收端就会把整包数据错位，把卡号完全解析错。

模块采用的 **7F 串口协议** 是一种兼具**帧同步**、**长度校验**、**多机寻址**、**功能区分**与**全段异或校验**的结构化数据协议。无论通信链路发生怎样的数据丢包或断线重连，接收端状态机都能精准锁定帧头，实现高可靠的数据解析。

## 2.2 协议帧解剖与格式规范

每一帧标准 7F 报文由以下 6 个逻辑部分紧密排列而成：




图 2-1　果蝇一号 DF-01 · 7F 串口通信协议帧结构解析

| 帧头 | LEN | ID | CMD | PARAM / DATA | XOR |
|---|---|---|---|---|---|
| `0x7F` | 长度 | 地址 | 命令 | 参数或数据负载 | 校验 |
| 1 字节 | 1 字节 | 1 字节 | 1 字节 | 0～252 字节，按具体命令定义 | 1 字节 |

1. **帧头（Header，1 字节）**：固定为十六进制数值 `0x7F`。作为整包报文的起始界定符。
2. **有效长度（LEN，1 字节）**：表示该帧在逻辑层面上紧随其后的字节总数。计算公式为：

$$\text{LEN} = \text{地址}(1B) + \text{命令}(1B) + \text{数据负载}(NB) + \text{校验和}(1B) = N + 3$$

3. **设备地址（ID，1 字节）**：模块在总线上的通信地址。范围为 `0x00` ~ `0xFE`，出厂默认地址为 `0x00`。在多机并联轮询系统中，可通过该字段区分不同的大门读卡器。
4. **命令码（CMD，1 字节）**：- **主机发送请求帧时**：CMD 的最高位为 0（如 `0x10` 读卡号、`0x11` 读块、`0x12` 写块）。- **模块回送响应帧时**：CMD 的最高位自动置 1，即 RESP_CMD = REQ_CMD | 0*x*80（如请求 `0x10`，应答必定为 `0x90`；请求 `0x11`，应答必定为 `0x91`）。这种对称设计让接收端判断回包类型极其轻松。
5. **参数与数据负载（PARAM / DATA，长度字段最多容纳 252 字节，具体命令按定义取值）**：- 请求帧中携带块号、密码、待写入数据等参数。- 响应帧中携带状态码 STATUS、卡片卡号 UID、16 字节数据块内容等。
6. **异或校验和（XOR Checksum，1 字节）**：用于保障数据在传输过程中不被干扰篡改。

## 2.3 核心亮点：0x7F 自动双写转义机制

仔细观察协议规则的工程师会提出一个经典问题：**"如果卡片里的数据或者计算出来的校验和正好就是 0x7F 怎么办？接收端岂不是会误把它当成新的一帧报文开头？"**

为了彻底解决这一问题，7F 协议引入了工业级**"0x7F 双写自动转义机制"**： - **发送端规则**：除了整帧第一个用来充当起始帧头的 `0x7F` 之外，报文后续任何字段（无论是长度、地址、命令、数据内容还是校验和），只要它的数值等于 `0x7F`，发送端硬件会自动连续发送两个 `0x7F`（即 `7F 7F`）。 - **接收端规则**：接收端状态机在收到起始 `0x7F` 之后，在后续解析过程中，若遇到连续两个 `0x7F`，会自动吃掉一个并还原为一个真正的 `0x7F`，并将其送入校验和计算队列。

> **实用技巧**
>
> **开发无忧**：在日常使用中，如果你使用的卡号或数据块没有恰好出现 0x7F，报文是原样传输的；而一旦出现 0x7F，本手册提供的 Python / C++ / Rust 驱动代码会自动透明完成双写与还原，开发者完全无需手动干预。

## 2.4 通俗异或校验和（XOR）怎么算?

异或校验（XOR）是嵌入式领域计算速度最快、检错能力极强的校验算法。它的数学特性是"相同为 0，不同为 1"，且具备自反性。

**计算范围**：从 `LEN` 字节开始，一直异或到数据负载的最后一个字节（帧头 `0x7F` 不参与）：

$$\mathrm{XOR} = \mathrm{LEN} \oplus \mathrm{ID} \oplus \mathrm{CMD} \oplus \mathrm{DATA}[0] \oplus \cdots \oplus \mathrm{DATA}[N-1]$$

### 手算实例：单次读卡号请求帧

- 请求帧：`7F 03 00 10 13`
- 参与校验的字节：`LEN = 03`、`ID = 00`、`CMD = 10`。
- 计算：`0x03 ^ 0x00 ^ 0x10 = 0x13` ✓

### Python 一行代码实现异或计算：

**Python**

```python
def calc_xor(logical_payload: bytes) -> int:
    """logical_payload = LEN + ID + CMD + PARAMS（帧头 0x7F 除外）"""
    chk = 0
    for b in logical_payload:
        chk ^= b
    return chk
```

## 2.5 状态码 STATUS 一览表

在所有模块返回的响应帧中，数据负载的第一个字节固定为 **操作状态码（STATUS）**。开发者在处理回包时，首先判断该字节是否为 `00` 即可得知操作成败：

| 状态码 (HEX) | 含义 | 产生原因与排查建议 |
| --- | --- | --- |
| `00` | **操作成功** | 卡片读写、配置修改或寻卡完全正常 |
| `FF` | **未寻到卡片** | 感应区无卡，或卡片材质不兼容、卡片已离开射频感应场 |
| `FE` | **执行失败** | 参数非法、选卡/读写失败、或内部非易失存储写入校验未通过 |
| `FD` | **密钥认证失败** | 目标扇区密码与模块当前装载的 Key A/B 不一致，被卡片拒绝 |

**说明与背景**

模块对校验和错误或地址不匹配的帧**直接静默丢弃**，不发回任何错误帧。因此收不到回包时，首先排查接线交叉（TX↔RX 互联）与校验和计算，而不是等待错误状态码。

---

## 2.6 全自动工作模式（Auto Mode）与业务分发

果蝇一号具备卓越的自主作业能力，内置了 4 种工作模式，通过指令 `2E` 进行配置并永久保存在内部存储中：




图 2-2　果蝇一号 DF-01 · 全自动寻卡与业务处理工作流程

| 模式码 (HEX) | 模式名称 | 行为特征与典型应用场景 |
| --- | --- | --- |
| `00` | 自动读取卡号 UID（Auto UID） | 刷卡后自动寻卡并读取 UID，无需密码认证，立即上报；延时极低（< 15 ms），适用于门禁、考勤与签到。 |
| `01` | 关闭自动读取（Manual Only） | 模块平时完全静默，只等待主机下发指令才执行动作；适用于严苛省电或主机严格定时查询的工业控制系统。 |
| `02` | 自动读取数据块（Auto Block） | 自动使用当前 Key A/B 认证目标扇区，读取 16 字节块数据后主动推送；适用于一卡通钱包消费、电子标签。 |
| `03` | 语音读块联动（Voice U13T） | 出厂默认模式。读取块数据后格式化为 U13T 语音帧，直接送出串口，驱动离线语音播报芯片发音。 |

## 2.7 硬件级防重读机制（Anti-Reread）

### 为什么必须要有防重读?

在现实生活中，用户刷卡后通常习惯将卡片贴在读卡器上停留 1~3 秒甚至更久。如果是普通的读卡芯片，每秒会向主机灌入 50~100 次相同的卡号数据，导致主机数据库疯狂插入重复记录、门禁继电器噼里啪啦反复吸合。

### 果蝇一号的优雅解法:

果蝇一号内置了硬件级独立防重读定时器（支持通过指令 `2F` 配置）： - **配置为 0 ms（出厂默认 · 单次锁定模式）**：卡片贴上后，模块仅读取并上报**一次**，指示灯点亮 500 ms。只要这张卡片一直停留在天线上，模块坚决保持静默；直到卡片彻底离开天线感应场、感应场净空重置后，下一次贴卡才会产生新的读卡事件。 - **配置为指定时长（如 500 ms、1000 ms、2000 ms）**：卡片停留期间，模块在设定的冷却时间内不重复上报，超过该时间若卡片仍在感应区，才会上报下一次。

---

## 2.8 射频增益（RF Gain）调谐技巧

果蝇一号支持 8 档射频接收电路增益调节（档位 `00H` ~ `07H`，对应接收灵敏度 18 dB ~ 48 dB）：

| 档位 | 对应增益 | 适用场景与调优指引 |
| --- | --- | --- |
| `07` | **48 dB** | **出厂默认档位。**最大接收灵敏度，标准塑料外壳或自由空间下感应距离最远（可达 55 mm） |
| `05` ~ `06` | 38~43 dB | 适用于天线贴近金属结构件、显示屏后壳或存在较强开关电源纹波干扰的环境，降低误码率 |
| `02` ~ `04` | 28~33 dB | 适用于紧凑型桌面机具，故意限制过远的感应距离，防止邻近手持卡片"误刷"或隔空误触 |
| `00` ~ `01` | 18~23 dB | 极近距离贴卡场景（< 15 mm），有效防止多卡堆叠串读 |

---

## 2.9 一键读取模块全参数（B1 帧结构全面拆解）

通过向模块发送指令 `7F 03 00 31 32`，模块会一次性将当前保存在内部存储中的所有运行参数全景回送（ `B1` 帧，共 34 字节参数）。

**报文 / 文本**

出厂默认参数回包示例：

```
7F 25 00 B1 00 00 00 C2 01 00 03 04 01 00 00 00 FF FF FF FF FF FF
FF FF FF FF FF FF 00 00 07 C4 09 E8 03 00 3C 00 4C
```

| 字节偏移 | 长度 | 字段名称 | 示例数据 (HEX) | 实际解析含义 |
| --- | --- | --- | --- | --- |
| **0** | 1 | `status` | `00` | 执行状态正常 |
| **1** | 1 | `module_id` | `00` | 当前模块地址为 `00` |
| **2** | 4 | `baud_rate` | `00 C2 01 00` | 小端序：0x0001C200 = **115200 bps** |
| **6** | 1 | `auto_mode` | `03` | 当前工作模式：模式 `03`（语音读块联动） |
| **7** | 1 | `auto_block` | `04` | 自动读取目标块号：第 4 块 |
| **8** | 4 | `auto_value` | `01 00 00 00` | 语音参数：GBK 编码，播放不阻塞 |
| **12** | 6 | `key_a` | `FF FF FF FF FF FF` | 当前保存的扇区密钥 A |
| **18** | 6 | `key_b` | `FF FF FF FF FF FF` | 当前保存的扇区密钥 B |
| **24** | 2 | `repeat_ms` | `00 00` | 防重读时间：0（单次感应锁定） |
| **26** | 1 | `gain` | `07` | 射频接收增益：档位 7（最大灵敏度 48 dB） |
| **27** | 2 | `ramp_ms` | `C4 09` | 小端序：0x09C4 = **2500 ms**（辅助输出缓变时间） |
| **29** | 2 | `delay_ms` | `E8 03` | 小端序：0x03E8 = **1000 ms**（系统启动延时） |
| **31** | 1 | `product_mode` | `00` | 产品模式：`00` 表示果蝇模式（只读 RFID，不启动 PWM 输出） |

| 字节偏移 | 长度 | 字段名称 | 示例数据 (HEX) | 实际解析含义 |
| --- | --- | --- | --- | --- |
| **32** | 1 | `initial_duty` | `3C` | 辅助输出初始占空比：0x3C = 60% |
| **33** | 1 | `motor_dir` | `00` | 辅助输出通道方向：通道 A |

> **说明与背景**
>
> 旧版固件返回的 `B1` 帧可能较短（27/31/32/33 字节），按实际 LEN 字段解析即可，缺失字段保持出厂默认值。

---

CHAPTER

# 第 3 章：配套工作台 DF-01 Studio 深度使用教程

## 3.1 跨平台工作台概览（免安装，即开即用）

为让开发者零门槛体验与调试果蝇一号，官方开发了轻量级、跨平台的图形化工作台 **DF-01 Studio**。该软件采用现代前端架构打造，无需安装任何复杂的底层驱动包，双击即可运行于 Windows、macOS 与 Linux 系统。




图 3-1　DF-01 Studio 官方可视化读卡工作台

## 3.2 模拟模式与串口实物连接

DF-01 Studio 创新性地设计了**双工作引擎**： 1. **实物串口模式（Hardware Mode）**：将果蝇一号通过 USB-TTL 连入电脑，在软件顶部的串口下拉框中选择对应 COM 口，选择波特率 115200，点击"连接设备"。软件将自动与硬件握手，并在界面右上角点亮绿色的"设备已连接"状态标。 2. **模拟设备模式（Simulator Mode）**：如果此时硬件模块尚未到货或不在手边，直接选择"模拟设备"，工作台会载入内置的高仿真虚拟果蝇一号内核。点击右侧的"模拟卡片在场"开关，即可仿真刷卡、读写扇区数据与参数配置，极大地便利了纯软件开发者的前期接口联调。

## 3.3 读卡工作台核心功能

工作台主界面划分为直观的功能卡片区： - **卡片识别卡片**：大字号即时展示当前刷卡卡号（HEX 小端与 DEC 十进制），同时显示卡片底层 ATQA 响应与最近一次读卡时间戳。 - **自动模式配置卡片**：一键切换"自动读卡号 (00)"、"关闭自动 (01)"、"自动读数据块 (02)"与"语音读块 (03)"，输入目标块号后点击"应用模式"，模块即时写入参数并永久生效。 - **射频增益滑块**：图形化滑块在 18 dB ~ 48 dB 之间无级拖动，点击"保存增益"即刻生效。 - **防重读时间预设**：提供"无限制(单次锁定)"、"500 ms"、"1000 ms"等快捷按钮，亦支持毫秒级自定义输入。 - **实时通信终端（Terminal）**：下半部分展示收发报文瀑布流，发送帧（TX 蓝色）与接收帧（RX 橙色）带时间戳详细区分，支持报文搜索、方向过滤与一键清屏。

## 3.4 多编码文本与数据解析工具

在 IC 卡开发中，很多开发者最头疼的就是 16 字节数据块的编码转换（例如：十六进制换算成汉字到底是用 UTF-8 还是 GBK？尾部的 `0x00` 结束符如何过滤？）。

DF-01 Studio 在右侧设计了**即时数据解码面板**： - 刷卡读取到 16 字节数据后，工作台会并行计算并同步展示 **HEX（十六进制）**、**UTF-8 字符**、**GBK 中文字符**、**GB18030**、**ASCII**、**UTF-16** 以及 **Base64**。 - 支持勾选“隐藏尾部 NUL”复选框，自动去除卡片扇区未占满位置补齐的 `0x00`，所见即所得。

---

## 3.5 密钥管理与卡片格式化

在“密钥管理”标签页中： - 预置了出厂通用密钥（`FF FF FF FF FF FF`）、交通系统密钥、门禁常用字典。 - 允许用户对整张卡片的 16 个扇区执行“一键认证扫描”、“批量密码修改”以及“初始化清零”。

---

## 3.6 四套专业设计主题切换

工作台右上角提供了 4 套精心设计、符合人体工学的主题皮肤：

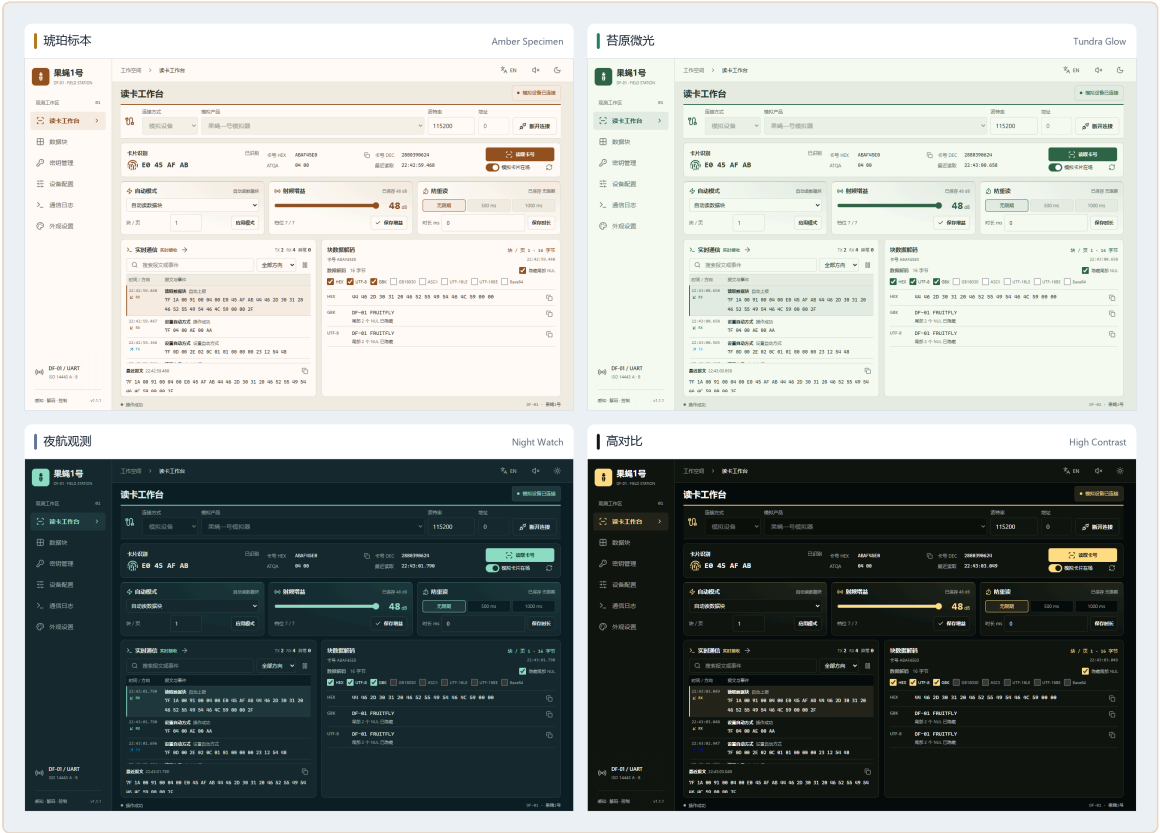



图 3-2　DF-01 Studio 四套主题配色：琥珀标本 / 苔原微光 / 夜航观测 / 高对比

- **琥珀标本（Amber Specimen）**：温润素雅的暖色纸质风格，适合日间室内办公。
- **苔原微光（Tundra Glow）**：低饱和度的青绿色调，清新护眼。

- **夜航观测（Night Watch）**：深邃的工业暗色系，适合夜间调试与弱光车间环境。
- **高对比（High Contrast）**：纯黑背景与高明度黄白文字，在户外强光或高对比度要求场景下提供最佳可读性。

---

CHAPTER

# 第 4 章：实战应用与代码复用（从零开发你的读卡项目）

## 4.0 示例源码与压缩包下载

本章的 Python、C/C++ 示例提供实际源码文件，可直接加入自己的项目。库代码与演示入口分开维护，无需从手册复制整段驱动代码。

| 下载包 | 内容 | 适用方式 |
| --- | --- | --- |
| Python 文件模块与示例 | `df01.py`、离线演示、串口示例、测试与 README | 复制文件后直接 `import`，无需 wheel |
| C / C++ 库与示例 | C 公共头文件与实现、C++ 封装、CMake、PC 串口与离线示例、测试 | 包含头文件并编译链接 C 实现 |
| 全部示例源码 | Python 与 C/C++ 完整配套源码及接入说明 | 按语言选用，或整体保存 |

HTML 版已内嵌以上三个压缩包，保存 HTML 单文件后也能离线点击下载。Markdown、DOCX、PDF 版请与 `downloads/` 文件夹一起保存；包名分别为 `df01-python-demo.zip`、`df01-c-cpp-demo.zip`、`df01-demo-all.zip`，校验值见 `downloads/SHA256SUMS.txt`。解压包内均包含 README 和可运行示例，不包含 wheel、编译器或可执行文件。

源码位于 `demos/df01/`，更新后运行 `python demos/build_packages.py` 重新生成压缩包，再运行三个手册构建脚本生成 HTML、DOCX、PDF。

## 4.1 通用 Python 驱动模块（可作为库直接导入）

要求 Python 3.10+。解压 Python 包，将 `df01.py` 放到自己的脚本同目录即可使用。帧编解码仅使用标准库；连接真实串口时才需要 `python -m pip install pyserial`。

**先运行无需硬件的演示**（在 Python 包解压目录执行）：

**Shell**

```
python offline.py
python df01.py --selftest
python -m unittest -v test_df01
```

离线演示输出请求帧 `7F 03 00 10 13`，并把示例回包按单字节分片送入解析器，输出卡号 `E0 45 AF AB` 和小端十进制 `2880390624`。

**作为文件模块导入：**

**Python**

```python
from df01 import DF01Client, FrameParser, encode_frame, parse_card

# 不打开串口也能使用帧编解码。
wire = encode_frame(0x00, 0x10)
print(wire.hex(" ").upper())

parser = FrameParser()
reply = bytes.fromhex("7F 0A 00 90 00 04 00 E0 45 AF AB 3F")
for frame in parser.feed(reply):
    card = parse_card(frame)
    print(card.uid_hex, card.uid_dec_str)
```

**连接真实串口并主动读卡：**

**Python**

```python
from df01 import DF01Client

with DF01Client("COM3", baud=115200, address=0x00) as client:
    print(client.get_config())
    if not client.disable_auto():
        raise RuntimeError("切换手动模式失败")
    # 模式切换前若已有自动上报，应先处理已有数据并等待链路空闲。
    card = client.read_uid(timeout=5.0)
    if card is not None:
        print(card.uid_hex, card.uid_dec_str)
```

`with` 退出会关闭串口。`disable_auto()` 会保存手动模式；如需恢复自动 UID 上报，调用 `set_auto_uid()`。串口完整示例 `example.py` 提供以下入口：

**Shell**

```
python example.py --port COM3
python example.py --port COM3 --block 4
python example.py --port COM3 --listen
python example.py --port /dev/ttyUSB0 --baud 115200 --address 0x00
```

主动读取示例保存手动模式，`--listen` 保存自动 UID 模式并持续打印卡号，Ctrl+C 退出。默认不写卡。

| 接口 | 行为 |
| --- | --- |
| `encode_frame()` / `FrameParser.feed()` | 编码线上字节 / 消费任意分片，产出有效帧迭代器 |
| `read_uid()` / `read_block(block)` | 成功返回 `CardInfo`，等待超时返回 `None`，模块失败状态抛 `DeviceError` |
| `get_config()` | 返回配置字典或超时 `None`，包含 B1 扩展字段，历史回包缺失字段为 `None` |
| `write_block(block, data)` / `load_keys(key_a, key_b)` | 写入 16 字节 / 装载两组 6 字节密钥，返回是否成功 |
| `set_auto_uid()` / `set_auto_block(block)` / `disable_auto()` | 自动 UID / 自动读块 / 手动模式，配置会保存 |
| `set_anti_repeat(ms)` / `set_gain(level)` | 防重读 0 或 100..3000 ms / 增益档位 0..7，非法值抛 `ValueError` |
| `listen(callback, stop_event=None)` | 接收自动上报并回调 `CardInfo`，回调异常交给应用处理 |
| `request(command, params=(), ...)` | 取得原始 `Frame`，调用者自行检查 `status` |

Python `Frame.status` 是状态字，`Frame.params` 不含状态字。可用 `DF01Client(transport=link)` 对接自己的串口对象：`read(n)` 最多返回 n 字节且必须有有限超时，`write(bytes)` 返回实际写入数，`close()` 关闭连接；客户端负责关闭该对象。完整约定见包内 README。

**重要提示**

**请求与自动上报的边界**

- 自动与主动回包共用 `90/91`，协议没有序列号。需要主动读卡时，先关闭自动模式、处理已有接收数据并等待空闲；不要同时运行 `listen()` 和同步请求，同一链路只允许一条请求在途。
- 当前固件无卡、读卡或读块失败时可能静默并保留单次请求。本地超时不会取消模块请求，之后放卡可能收到迟到帧；请先处理迟到帧并确认状态，再发起下一项业务。
- UID 为固件回送的 4 字节；长 UID 卡回送末 4 字节。`uid_hex` 按线上顺序显示，十进制按小端解释。

## 4.2 C / C++ 驱动库（包含头文件并链接实现）

C11 核心不依赖操作系统、第三方串口库或堆内存。C++11 封装复用同一个 C 实现，避免两套协议代码产生差异。

| 文件 | 用途 |
| --- | --- |
| `include/df01.h` + `src/df01.c` | C 公共 API 与协议实现，适合 MCU 和 PC |
| `include/df01.hpp` | C++ 的 `Encoder`、`Parser`、卡片与配置解析封装 |
| `examples/demo.c` / `examples/demo.cpp` | 可直接编译运行的 C / C++ 离线演示 |
| `examples/serial.c` | Windows / POSIX 串口配置回读示例，固定地址 00、115200 8N1 |
| `tests/test_protocol.c` | 长度边界、转义、校验、残帧恢复与回包解析验证 |

**构建与运行**（在 C/C++ 包解压目录执行）：

**Shell**

```
cmake -S . -B build
cmake --build build --config Release
ctest --test-dir build -C Release --output-on-failure
```

单配置生成器的程序位于 `build/`；Visual Studio 等多配置生成器的程序位于 `build/Release/`。Windows MinGW 可使用 `-G "MinGW Makefiles"`；若工具链不支持中文路径，请解压到英文目录。

**Shell**

```
./build/df01_c_demo
./build/df01_cpp_demo
./build/df01_serial /dev/ttyUSB0
```

Windows 串口例：`.\build\df01_serial.exe COM3`。串口示例只读取配置，不修改模式；macOS 可换成 `/dev/cu.*`。Windows 分支已编译，POSIX 分支需在目标平台构建验收；未连接实物测试。

**C 工程接入**：把 `src/df01.c` 加入源文件，将 `include/` 加入头文件路径。

**C**

```c
#include "df01.h"

void prepare_read_uid(void)
{
    uint8_t wire[DF01_MAX_WIRE];
    size_t length = df01_encode(0, DF01_READ_UID, NULL, 0, wire, sizeof(wire));
```

```
    /* length 为 0 表示失败，否则将 wire 的前 length 字节交给自己的 UART。 */
    (void)length;
}
```

接收端先 `df01_decoder_reset(&decoder)`，再逐字节调用 `df01_decode_byte(&decoder, byte, &frame)`；返回 1 时才处理帧。先检查 `frame.address` 和 `frame.command`，再调用 `df01_parse_card()` 或 `df01_parse_config()`。完整可运行接收流程见 `examples/demo.c`。

**C++ 工程接入**：包含 `df01.hpp`，同时编译链接 `df01.c`。

**C++**

```cpp
#include "df01.hpp"

void prepare_read_uid()
{
    df01::Encoder encoder{0x00};
    df01::Parser parser;
    uint8_t wire[df01::MAX_WIRE];
    size_t length = encoder.encode(df01::CMD_READ_UID, wire, sizeof(wire));
    // 发送 wire[0..length)，接收字节交给 parser.feed(byte, &frame)。
    (void)length;
}
```

如果项目使用 CMake，可直接导入库目标：

**CMake**

```cmake
add_subdirectory(path/to/native df01-build)
target_link_libraries(your_app PRIVATE df01::df01)
```

作为子目录时默认不构建 PC 示例，也不强制启用 C++ 工具链。单独构建仅需要库时，可传入 `-DDF01_BUILD_EXAMPLES=OFF -DBUILD_TESTING=OFF`。

> **说明与背景**
>
> C/C++ `frame.params` 包含原始 STATUS，`param_count` 也包含该字节。`df01_parse_card()` / `df01_parse_config()` 返回 1 才表示完整成功响应；失败时请检查原始状态和长度。配置解析支持当前 34 字节 B1 回包，历史缺失字段清零，并用 `config.param_count` 区分缺失与零值。

---

## 4.3 Rust 驱动库（no_std 兼容，可用于嵌入式与 PC）

将以下代码保存为 `df01.rs`，在项目中 `mod df01;` 后即可使用。`no_std` 模式下去掉 `std` feature 即可在嵌入式环境运行。

**Rust**

```rust
//! df01.rs — DF-01 通用 Rust 驱动库
//!
//! # 用法
//! ```rust
//! use df01::{Encoder, Parser, Frame};
//!
//! let enc = Encoder::new(0x00);
//! let mut buf = [0u8; df01::MAX_WIRE];
//! let len = enc.encode_read_uid(&mut buf);
//! // serial_write(&buf[..len]);
//!
//! let mut parser = Parser::new();
//! // for byte in serial_read_iter() { if let Some(f) = parser.feed(byte) { ... } }
//! ```

#![allow(dead_code)]

pub const HEADER: u8 = 0x7F;

pub const ST_OK:      u8 = 0x00;
pub const ST_NO_CARD: u8 = 0xFF;
pub const ST_ERROR:   u8 = 0xFE;
pub const ST_PWDERR:  u8 = 0xFD;

pub const CMD_READ_UID:    u8 = 0x10;
pub const CMD_READ_BLOCK:  u8 = 0x11;
pub const CMD_WRITE_BLOCK: u8 = 0x12;
pub const CMD_LOAD_KEY:    u8 = 0x2B;
pub const CMD_SET_BAUD:    u8 = 0x2C;
pub const CMD_SET_ID:      u8 = 0x2D;
pub const CMD_SET_AUTO:    u8 = 0x2E;
pub const CMD_SET_REPEAT:  u8 = 0x2F;
pub const CMD_SET_GAIN:    u8 = 0x30;
pub const CMD_GET_CONFIG:  u8 = 0x31;

pub const MAX_PARAMS: usize = 24;
pub const MAX_WIRE:   usize = 1 + (MAX_PARAMS + 4) * 2;

// ---------------------------------------------------------------- 帧结构

#[derive(Debug, Clone)]
pub struct Frame {
    pub address:     u8,
    pub command:     u8,
    pub status:      u8,
    pub params:      [u8; MAX_PARAMS],
    pub param_count: u8,
}

impl Frame {
    /// 从 0x90 / 0x91 帧提取 4 字节 UID（小端）。
    pub fn uid(&self) -> Option<[u8; 4]> {
        if self.param_count < 6 { return None; }
        Some([self.params[2], self.params[3], self.params[4], self.params[5]])
    }
```

```rust
    /// UID 十进制（小端 u32）。
    pub fn uid_dec(&self) -> Option<u32> {
        let u = self.uid()?;
        Some(u32::from_le_bytes(u))
    }

    /// 16 字节块数据（仅 0x91 帧）。
    pub fn block_data(&self) -> Option<&[u8]> {
        if self.param_count < 22 { return None; }
        Some(&self.params[6..22])
    }
}

// ---------------------------------------------------------------- 编码器

pub struct Encoder {
    address: u8,
}

impl Encoder {
    pub const fn new(address: u8) -> Self {
        Self { address }
    }

    /// 将命令编码为线上字节串，写入 `out`，返回字节数。
    pub fn encode(&self, command: u8, params: &[u8], out: &mut [u8]) -> usize {
        assert!(out.len() >= MAX_WIRE);
        let length = (3 + params.len()) as u8;
        let mut xor = length ^ self.address ^ command;
        for &b in params { xor ^= b; }

        let mut logical = [0u8; MAX_PARAMS + 4];
        logical[0] = length;
        logical[1] = self.address;
        logical[2] = command;
        logical[3..3 + params.len()].copy_from_slice(params);
        logical[3 + params.len()] = xor;

        let mut w = 0usize;
        out[w] = HEADER; w += 1;
        for i in 0..=(length as usize) {
            out[w] = logical[i]; w += 1;
            if logical[i] == HEADER { out[w] = HEADER; w += 1; }  // 0x7F 双写转义
        }
        w
    }

    pub fn encode_read_uid(&self, out: &mut [u8]) -> usize {
        self.encode(CMD_READ_UID, &[], out)
    }

    pub fn encode_read_block(&self, block: u8, out: &mut [u8]) -> usize {
        self.encode(CMD_READ_BLOCK, &[block], out)
    }

    pub fn encode_write_block(&self, block: u8, data: &[u8; 16], out: &mut [u8]) -> usize {
```

```
        let mut p = [0u8; 17];
        p[0] = block;
        p[1..].copy_from_slice(data);
        self.encode(CMD_WRITE_BLOCK, &p, out)
    }

    pub fn encode_set_gain(&self, level: u8, out: &mut [u8]) -> usize {
        self.encode(CMD_SET_GAIN, &[level & 0x07], out)
    }

    pub fn encode_set_anti_repeat(&self, ms: u16, out: &mut [u8]) -> usize {
        self.encode(CMD_SET_REPEAT, &[(ms & 0xFF) as u8, (ms >> 8) as u8], out)
    }

    pub fn encode_set_auto_uid(&self, out: &mut [u8]) -> usize {
        self.encode_set_auto(0x00, 4, out)
    }

    pub fn encode_set_auto_block(&self, block: u8, out: &mut [u8]) -> usize {
        self.encode_set_auto(0x02, block, out)
    }

    pub fn encode_disable_auto(&self, out: &mut [u8]) -> usize {
        self.encode_set_auto(0x01, 4, out)
    }

    fn encode_set_auto(&self, mode: u8, block: u8, out: &mut [u8]) -> usize {
        let params = [mode, mode.wrapping_add(0x0A), block,
                      0x01, 0x00, 0x00, 0x00, 0x23, 0x12, 0x54];
        self.encode(CMD_SET_AUTO, &params, out)
    }

    pub fn encode_get_config(&self, out: &mut [u8]) -> usize {
        self.encode(CMD_GET_CONFIG, &[], out)
    }
}

// ---------------------------------------------------------------- 解析器

#[derive(Default)]
enum State { #[default] Idle, Active, Escape }

pub struct Parser {
    raw:    [u8; MAX_PARAMS + 5],
    count:  usize,
    state:  State,
}

impl Parser {
    pub const fn new() -> Self {
        Self { raw: [0; MAX_PARAMS + 5], count: 0, state: State::Idle }
    }

    /// 喂入一字节，完整有效帧时返回 `Some(Frame)`，否则返回 `None`。
    pub fn feed(&mut self, byte: u8) -> Option<Frame> {
        match self.state {
            State::Idle => {
```

```
                if byte == HEADER {
                    self.count = 0;
                    self.state = State::Active;
                }
                None
            }
            State::Active => {
                if byte == HEADER {
                    self.state = State::Escape;
                    return None;
                }
                self.push(byte)
            }
            State::Escape => {
                if byte == HEADER {
                    // 0x7F 0x7F → 一个 0x7F
                    self.state = State::Active;
                    self.push(HEADER)
                } else {
                    // 单独 0x7F 后跟别的：重新开始
                    self.count = 0;
                    self.state = State::Active;
                    self.push(byte)
                }
            }
        }
    }

    fn push(&mut self, byte: u8) -> Option<Frame> {
        if self.count < self.raw.len() {
            self.raw[self.count] = byte;
            self.count += 1;
        }
        if self.count < 1 { return None; }
        let length = self.raw[0] as usize;
        if length < 3 || length > MAX_PARAMS + 3 {
            self.count = 0;
            self.state = State::Idle;
            return None;
        }
        if self.count < length + 1 { return None; }

        // 校验
        let xsum = self.raw[..length].iter().fold(0u8, |a, &b| a ^ b);
        let stored = self.raw[length];
        self.count = 0;
        self.state = State::Idle;
        if xsum != stored { return None; }

        let mut frame = Frame {
            address:     self.raw[1],
            command:     self.raw[2],
            status:      if length >= 4 { self.raw[3] } else { 0 },
            params:      [0; MAX_PARAMS],
            param_count: if length >= 4 { (length - 4) as u8 } else { 0 },
        };
        let pc = frame.param_count as usize;
```

```rust
        frame.params[..pc].copy_from_slice(&self.raw[4..4 + pc]);
        Some(frame)
    }
}

// ---------------------------------------------------------------- 自检

#[cfg(test)]
mod tests {
    use super::*;

    fn hex(frame: &[u8]) -> String {
        frame.iter().map(|b| format!("{:02X}", b)).collect::<Vec<_>>().join(" ")
    }

    #[test]
    fn encode_read_uid() {
        let enc = Encoder::new(0x00);
        let mut buf = [0u8; MAX_WIRE];
        let n = enc.encode_read_uid(&mut buf);
        assert_eq!(hex(&buf[..n]), "7F 03 00 10 13");
    }

    #[test]
    fn encode_anti_repeat_1000ms() {
        let enc = Encoder::new(0x00);
        let mut buf = [0u8; MAX_WIRE];
        let n = enc.encode_set_anti_repeat(1000, &mut buf);
        assert_eq!(hex(&buf[..n]), "7F 05 00 2F E8 03 C1");
    }

    #[test]
    fn escape_7f_in_params() {
        let enc = Encoder::new(0x00);
        let mut buf = [0u8; MAX_WIRE];
        let n = enc.encode(CMD_SET_GAIN, &[0x7F], &mut buf);
        assert_eq!(hex(&buf[..n]), "7F 04 00 30 7F 7F 4B");
    }

    #[test]
    fn roundtrip() {
        let enc = Encoder::new(0x00);
        let mut buf = [0u8; MAX_WIRE];
        let n = enc.encode_read_uid(&mut buf);
        let mut parser = Parser::new();
        // 模拟模块回包：7F 0A 00 90 00 04 00 E0 45 AF AB 3F
        let reply = [0x7Fu8, 0x0A, 0x00, 0x90, 0x00, 0x04, 0x00,
                     0xE0, 0x45, 0xAF, 0xAB, 0x3F];
        let _ = n;
        let mut got: Option<Frame> = None;
        for &b in &reply { got = parser.feed(b).or(got); }
        let f = got.unwrap();
        assert_eq!(f.command, 0x90);
        assert_eq!(f.status, ST_OK);
        assert_eq!(f.uid_dec(), Some(0xABAF45E0));
```

```
    }
}
```

## 4.4 四大经典工程落地配方

### 配方 1：智能门禁考勤（自动读 UID + 防重读 1s + 开锁继电器）

- **配置步骤**： 1. 向模块发送 `7F 0D 00 2E 00 0A 04 00 00 00 00 23 12 54 48` 将工作模式设为模式 00（自动读 UID）。 2. 向模块发送 `7F 05 00 2F E8 03 C1` 将防重读时间设置为 1000 ms。
- **硬件联动**：将第 5 引脚 `LED` 经 NPN 三极管（基极串 1kΩ 电阻）驱动 12V 门禁电磁锁继电器线圈。
- **运行表现**：用户刷卡瞬间，板载灯亮，外部继电器自动吸合 500 ms 释放门锁；串口同时将卡号推送给主控做权限校验；用户长按不拿开绝不发生二次打卡。

### 配方 2：会员卡储值与消费扣款（扇区密码校验 + 块数据读写）

- **业务流程**： 1. 使用指令 `2B` 装载商户专用的 6 字节扇区密钥（如 `A0 A1 A2 A3 A4 A5`）。 2. 卡片靠近后，通过指令 `11` 读取钱包块（如第 4 块）16 字节数据。 3. 主控校验块内余额字段；扣除消费金额后，调用指令 `12` 将最新余额写回卡片。 4. 读卡器返回成功状态码 `00`，完成闭环消费。

### 配方 3：智能工具柜与借还盘点

- 在工具标签中写入工具编号、归属部门与校验哈希。
- 借还人员将工具在模块上方掠过，通过自动读块模式瞬时解析出工具信息并在管理屏上登记录入，盘点效率较条码扫码枪提升数倍。

### 配方 4：工业现场多机总线（RS-485 多读卡器轮询组网）

- **多机并联**：在大型车间有 8 个工位需要刷卡，可通过外部 RS-485 收发器将模块的 TTL 串口转换为差分总线。
- **地址分配**：使用指令 `2D` 分别将各模块地址配置为 `01`、`02` … `08`。
- **轮询查询**：将所有模块设为模式 01（关闭自动上报），由工控机总线依次向各地址发送 `7F 03 [ID] 10 [XOR]` 定时巡检，彻底杜绝总线数据冲突。

---

CHAPTER

# 第 5 章：完整指令参考手册（开发者权威字典）

## 5.1 指令一览总表

| 请求 CMD | 响应 CMD | 指令功能描述 | 数据参数长度 | 是否保存至非易失存储 | 生效时机 |
|---|---|---|---|---|---|
| `10` | `90` | 单次寻卡并读取卡号 (UID) | 0 字节 | 否 | 立即执行一次 |
| `11` | `91` | 单次读取指定扇区数据块 | 1 字节（块号） | 否 | 立即执行一次 |
| `12` | `92` | 写入指定扇区数据块 (16B) | 17 字节（块号+16B数据） | 否 | 立即执行一次 |
| `2B` | `AB` | 装载扇区密码 (Key A / Key B) | 18 字节 | **是** | 立即生效并保存 |
| `2C` | `AC` | 修改串口通信波特率 | 7 字节 | **是** | 立即生效（发送完回包后切换） |
| `2D` | `AD` | 修改模块设备地址 (Device ID) | 4 字节 | **是** | 立即生效（应答使用新地址） |
| `2E` | `AE` | 配置自动工作模式 (Auto Mode) | 10 字节 | **是** | 立即生效并保存 |
| `2F` | `AF` | 配置防重读冷却时间 | 2 字节 | **是** | 立即生效并保存 |
| `30` | `B0` | 配置射频接收电路增益档位 | 1 字节 | **是** | 立即生效并保存 |
| `31` | `B1` | 一键回读模块全部配置参数 | 0 字节 | 否 | 立即返回 34 字节参数 |
| `32` | `B2` | 配置辅助输出缓变时间与上电延时 | 4 字节 | **是** | 下次上电生效 |

| 请求 CMD | 响应 CMD | 指令功能描述 | 数据参数长度 | 是否保存至非易失存储 | 生效时机 |
|---|---|---|---|---|---|
| `33` | `B3` | 配置辅助输出初始电平/占空比 | 1 字节 | **是** | 下次上电生效 |
| `34` | `B4` | 配置辅助输出通道模式与极性 | 1 字节 | **是** | 下次上电生效 |

---

## 5.2 `10` / `90` 单次读取卡号 UID

主动触发模块执行一次寻卡操作，并返回卡片 UID 与卡型 ATQA。

- **请求帧**:

  **报文 / 文本**

  ```
  7F 03 00 10 13
  ```

- **成功响应帧**:

  **报文 / 文本**

  ```
  7F 0A 00 90 STATUS ATQA0 ATQA1 UID0 UID1 UID2 UID3 XOR
  ```

- `STATUS`：`00` 表示成功。
- `ATQA`：2 字节卡型特征（标准 MIFARE Classic 通常为 `04 00`）。
- `UID`：4 字节卡号（若是 7 字节或 10 字节长卡号，回送其末尾 4 字节）。
- **无卡或读取失败**：当前固件保持静默，并保留单次读卡请求直到成功上报。主机等待超时不会取消模块请求，之后贴卡可能收到迟到响应。

---

## 5.3 `11` / `91` 单次读取数据块

指定绝对块号，使用当前保存的 Key A 或 Key B 执行扇区密码认证，成功后读出整块 16 字节内容。

- **请求帧示例（读取第 4 块）**：

**报文 / 文本**

```
7F 04 00 11 04 11
```

- **成功响应帧**：

**报文 / 文本**

```
7F 1A 00 91 00 04 00 E0 45 AF AB [16 字节数据内容] XOR
```

- 响应帧中包含了状态码 `00`、卡型、4 字节 UID 以及完整的 16 字节块数据。

---

## 5.4 `12` / `92` 写入数据块

指定绝对块号，写入 16 字节新数据。

- **请求帧示例（向第 4 块写入 16 个 0x88）**：

**报文 / 文本**

```
7F 14 00 12 04 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 XOR
```

- **成功响应帧**：

**报文 / 文本**

```
7F 04 00 92 00 96
```

- `STATUS` 为 `00` 表示物理写入与回读验证全部通过。

**安全警告**

**绝对不要随意写第 0 块与控制块！**

- 第 0 块（块号 0）是厂家只读信息块，普通卡片写入会直接报写失败。
- 每个扇区的最后一块（如第 3、7、11、15...块）是"扇区控制块"，存放了 Key A、权限控制位和 Key B。如果写入了错误的控制位，该扇区将永久死锁报废！初学者请务必在常规数据块（如第 1、2、4、5、6 块）上进行写入实验。

---

## 5.5 `2B` / `AB` 装载扇区认证密钥 (Key A / Key B)

模块内部保存一组 6 字节 Key A 与一组 6 字节 Key B，用于 MIFARE Classic 扇区认证。出厂默认均为全 `FF`。

- **请求帧格式（18 字节参数）**：

| 偏移 | 长度 | 内容 |
|---|---|---|
| 0 | 6 | Key A |
| 6 | 6 | Key B |
| 12 | 6 | 固定确认码：`00 03 08 05 02 07` |

双 Key 均重置为默认值示例：

**报文 / 文本**

```
7F 15 00 2B FF FF FF FF FF FF FF FF FF FF FF FF 00 03 08 05 02 07 35
```

- **成功响应帧**：

  **报文 / 文本**

  ```
  7F 04 00 AB 00 AF
  ```

  保存成功后立即生效，之后所有认证操作均使用新密钥。参数长度不是 18 字节、确认码不符或存储写入失败时返回 `FE`。

---

## 5.6 `2C` / `AC` 修改工作波特率

支持将串口波特率修改为以下任一值：`2400`、`4800`、`9600`、`14400`、`19200`、`28800`、`38400`、`57600`、`115200 bps`。修改后保存至内部非易失存储，掉电不丢失。

- **请求帧格式（7 字节参数）**：

| 偏移 | 长度 | 内容 |
| --- | --- | --- |
| 0 | 4 | 目标波特率，小端序（也接受大端序输入） |
| 4 | 3 | 固定确认码：`98 24 31` |

切换到 115200 bps 示例：

**报文 / 文本**

```
7F 0A 00 2C 00 C2 01 00 98 24 31 68
```

- **成功响应帧（以当前旧波特率发送）**：

  **报文 / 文本**

  ```
  7F 04 00 AC 00 A8
  ```

  模块在发送完应答帧后才切换到新波特率，主机需同步更改串口设置。参数非法或存储失败时返回 `FE`，波特率不变。

---

## 5.7 `2D` / `AD` 修改模块设备地址 (ID)

将模块通信地址修改为 `00H` ~ `FFH`，适用于多台读卡器挂载在同一条总线上的场景。

- **请求帧格式（4 字节参数）**：`新地址(1B)` + 固定确认码 `37 21 56`

将地址改为 `01` 示例：

**报文 / 文本**

```
7F 07 00 2D 01 37 21 56 6B
```

- **成功响应帧（新地址即时生效）**：

  **报文 / 文本**

  ```
  7F 04 01 AD 00 A8
  ```

  响应帧中的 ID 字段已切换为新地址。确认码错误或存储失败时用原地址回 `FE`。

## 5.8 `2E` / `AE` 配置自动工作模式

- **请求帧格式（设为自动上报卡号 00）**：

| 报文 / 文本 |
| --- |
| `7F 0D 00 2E 00 0A 04 01 00 00 00 23 12 54 4D` |

- **请求帧格式（设为自动读第 4 块 02）**：

| 报文 / 文本 |
| --- |
| `7F 0D 00 2E 02 0C 04 01 00 00 00 23 12 54 4D` |

---

## 5.9 `2F` / `AF` 配置防重读时间

参数为 2 字节毫秒数（小端序）。`00 00` 表示单次感应锁定（出厂默认）；`E8 03` 表示 1000 ms。

- **请求帧示例（设置防重读 1000 ms）**：

| 报文 / 文本 |
| --- |
| `7F 05 00 2F E8 03 C1` |

- **响应帧**：

| 报文 / 文本 |
| --- |
| `7F 04 00 AF 00 AB` |

---

## 5.10 `30` / `B0` 配置射频接收增益档位

参数为 1 字节档位（`00` ~ `07`，对应 18 dB ~ 48 dB）。

- **请求帧示例（设为最大档 07）**：

| 报文 / 文本 |
| --- |
| `7F 04 00 30 07 33` |

- **响应帧**：

**报文 / 文本**

```
7F 04 00 B0 00 B4
```

---

## 5.11 `31` / `B1` 读取模块全部当前配置参数

- **请求帧**：

**报文 / 文本**

```
7F 03 00 31 32
```

- **响应帧**：回送 34 字节原始参数（LEN=25，未转义完整帧共 39 字节）（详细字段定义参见 2.9 节）。

---

## 5.12 `32` / `B2` 配置扩展输出缓变时间与上电延时

- 参数为 4 字节小端序（缓变时间 2B + 启动延时 2B），范围 200~10000 ms 与 200~5000 ms。
- 保存至内部存储，下次上电生效。

---

## 5.13 `33` / `B3` 配置扩展输出初始电平/占空比

- 参数为 1 字节百分比数值（`00` ~ `64`，即 0% ~ 100%）。默认值为 `3C`（60%）。
- 保存至内部存储，下次上电生效。

---

## 5.14 `34` / `B4` 配置扩展输出启动极性/通道模式

- 参数为 1 字节（`00` 或 `01`），切换扩展辅助输出通道的驱动极性。
- 保存至内部存储，下次上电生效。

---

## 5.15 预留指令说明

指令码 `13`（制卡）、`14`（清卡）、`15`（充值）、`16`（扣款）为历史预留指令，当前固件推荐直接在业务层通过 `11` 读块与 `12` 写块统一实现钱包与账户逻辑。

---

CHAPTER

# 第 6 章：工程化、验收与排障

## 6.1 进料与出厂检验（IQC / OQC 规范测试流程）

在批量采购与项目集成落地时，推荐按照以下规范流程进行进料检验：

1. **外观与排针检验**：PCB 表面无划伤、油墨均匀、排针平直无虚焊；天线线圈边缘无铜箔翘起。
2. **通电与自检静态电流**：接入稳压 3.3V/5V 直流电源，使用微安表或直流电源监测： - 静态电流应在 30 ~ 45 mA 之间。 - 板载 SYS 绿灯应在 100 ms 内稳定点亮。 - 板载 READ 读卡灯在上电自检时短暂微闪后熄灭。
3. **感应距离检验**： - 使用一张标准 MIFARE Classic 白卡，从模块正上方垂直靠近天线。 - 当距离达到 **40 ~ 50 mm** 时，READ 指示灯应清脆点亮，说明天线谐振频率与射频增益完全达标。
4. **串口报文与异或校验验证**：通过串口助手接收 7F 报文，核对卡号 UID 准确性，验证 XOR 校验位无误。
5. **参数持久化断电验证**：修改一次射频增益或防重读时间，断电重启后发送 `31` 指令回读参数，确认断电配置依然完好保存。

---

## 6.2 常见问题排障指南（FAQ 十问十答）

### Q1：模块插上电后，SYS 绿灯不亮？

- **排查步骤**： 1. 检查电源极性是否接反，VCC 与 GND 是否接触不良。 2. 测量模块引脚上的实际输入电压是否处于 3.0V ~ 5.5V 范围内。

### Q2：刷卡时 READ 蓝灯会亮，但串口调试助手没有任何数据输出？

- **排查步骤**： 1. **检查 TX/RX 是否接反**：模块的 TX 必须接 USB-TTL 的 RX，模块的 RX 必须接 USB-TTL 的 TX。 2. **检查串口参数**：波特率是否设为了 115200？数据位 8，停止位 1，无校验？ 3. **检查 COM 口号**：确认电脑设备管理器中是否选对了对应的 USB-TTL 串口号。 4. **共地检查**：如果是连接主控板，主控板的 GND 必须与模块的 GND 牢固连接在同一地线上。

### Q3：为什么有时候刷卡会收到一串 `FD` 开头的报文，而不是 `7F`？

- **解答**：这说明模块当前处于出厂默认的**模式 03（语音播报联动模式）**！模块读出数据块后将其自动打包为了标准 U13T 语音帧（以 `0xFD` 开头）。只需向模块发送指令 `7F 0D 00 2E 00 0A 04 00 00 00 00 23 12 54 48`，即可一键切换为标准的 **模式 00（自动推送 7F 卡号帧）**。

### Q4：读取扇区数据块返回密钥认证失败（`FD`）？

- **解答**：MIFARE 卡片出厂密码一般为 6 个 `0xFF`。如果卡片已被其他系统加密（例如被门禁系统修改过密码），模块用默认全 FF 密码就无法通过认证，卡片会直接拒绝读取。需使用指令 `2B` 将正确的 6 字节密码装载入模块后再试。

### Q5：为什么读取卡号一切正常，但向数据块写入数据时报错?

- **解答**： 1. 确认写入的不是第 0 块（厂家信息只读块）。 2. 确认写入的数据刚好为 **16 个字节**，不多也不少。 3. 确认写入的目标扇区密码认证权限允许写操作（部分卡片控制位被设置为只读）。

### Q6：为什么串口助手收到的报文校验和（XOR）对不上?

- **解答**：请检查报文中是否包含了连续的两个 `0x7F`（`7F 7F`）。由于 0x7F 转义机制的存在，串口物理层传输时会多出一个转义字节，接收端在计算校验和之前必须先将其还原为一个 `0x7F`。

### Q7：卡片贴在天线上拿不拿开，读卡灯只亮一次，之后再刷为什么不亮了?

- **解答**：这是防重读保护生效的正常现象。卡片必须离开天线感应场至少 100 ms 以上，让天线感应场彻底复位，下次靠近才会重新触发下一次刷卡。

### Q8：如何修改模块的通信波特率？万一改乱了忘记了怎么办?

- **解答**：使用指令 `2C` 可以修改波特率。支持的波特率：2400、4800、9600、14400、19200、28800、38400、57600、115200 bps。如果遗忘了配置好的波特率，可在串口助手中逐一尝试上述波特率，发送 `7F 03 00 31 32` 直到收到正确应答即可查明。

### Q9：LED 联动引脚可以直接接 12V 继电器吗?

- **解答**：**不能直接接 12V 电源！** LED 引脚为 3.3V 逻辑输出引脚。驱动 12V 继电器必须串联 1kΩ 电阻后接 NPN 三极管作为开关驱动，三极管集电极接继电器线圈，发射极接 GND，见第 1.3 节驱动电路图。

### Q10：感应距离感觉偏近，有什么优化方法?

- **解答**： 1. 确认指令 `30` 的射频增益档位已设为最大档 `07`（48 dB）。 2. 确保模块天线背面不要紧贴大面积铜箔或金属机壳。 3. 尽量使用高品质的标准 ISO 卡，微型异形扣因天线面积小感应距离天然会比大卡偏近 30% 左右。

---

## 6.3 硬件安装与外壳结构建议

1. **天线净空要求**：13.56 MHz 是高频交变磁场，金属物体会在磁场中产生涡流并极大地削弱磁通量。安装时，模块正上方与周围 **15 mm 范围内**严禁有大块金属板、金属螺丝柱或锂电池屏蔽；天线背面若必须紧靠金属底座，应在天线与金属板之间加贴 **厚度 0.3~0.5 mm 的铁氧体吸波隔磁片**。

2. **塑胶外壳厚度**：常见的 ABS、PC、PLA、亚克力等非金属外壳对 13.56MHz 磁场完全透明，外壳面板厚度建议控制在 **1.0 ~ 3.0 mm** 之间，以保留充裕的感应余量。
3. **走线抗干扰**：供电电源入口建议并联一颗 10 μF 钽电容和一颗 100 nF 贴片陶瓷去耦电容，滤除工业开关电源高频纹波。

CHAPTER

# 附录

## 附录 A：标准请求与应答报文速查表

| 功能描述 | 主机发送报文 (HEX) | 模块返回报文 (HEX) | 备注说明 |
|---|---|---|---|
| **单次读取卡号 UID** | `7F 03 00 10 13` | `7F 0A 00 90 00 [ATQA 2B] [UID 4B] XOR` | 出厂默认地址 ID=00 |
| **单次读第 4 块数据** | `7F 04 00 11 04 11` | `7F 1A 00 91 00 04 00 [UID 4B] [数据 16B] XOR` | 需已装载对应扇区密码 |
| **向第 4 块写 16 个 00** | `7F 14 00 12 04 00 00...00 XOR` | `7F 04 00 92 00 96` | 写入并回读验证通过 |
| **切换为自动读卡号** | `7F 0D 00 2E 00 0A 04 00 00 00 00 23 12 54 48` | `7F 04 00 AE 00 AA` | 设为模式 00，刷卡主动推 UID |
| **切换为自动读第 4 块** | `7F 0D 00 2E 02 0C 04 00 00 00 00 23 12 54 4C` | `7F 04 00 AE 00 AA` | 设为模式 02，刷卡主动推 16B |
| **关闭自动读取模式** | `7F 0D 00 2E 01 0B 04 00 00 00 00 23 12 54 48` | `7F 04 00 AE 00 AA` | 设为模式 01，仅被动响应 |
| **设置防重读 1000 ms** | `7F 05 00 2F E8 03 C1` | `7F 04 00 AF 00 AB` | 小端序：0x03E8 = 1000 ms |
| **设置防重读单次锁定** | `7F 05 00 2F 00 00 2A` | `7F 04 00 AF 00 AB` | 出厂默认：不拿开绝不重报 |
| **射频增益设为最大档** | `7F 04 00 30 07 33` | `7F 04 00 B0 00 B4` | 档位 07（48 dB 最大灵敏度） |
| **一键回读全部参数** | `7F 03 00 31 32` | `7F 25 00 B1 00 [参数 34B] XOR` | 返回 37 字节完整镜像 |

# 附录 B：MIFARE Classic 1K 存储结构全解

MIFARE Classic 1K 是全球应用最广泛的高频 IC 卡： - **总容量**：1024 字节（1 KB），划分为 **16 个扇区（Sector 0 ~ 15）**。 - **每个扇区结构**：包含 **4 个数据块（Block 0 ~ 3）**，每个块固定为 **16 个字节**。 - **绝对块号计算**：

$$绝对块号 = 扇区号 \times 4 + 扇区内块偏移$$

（例如：扇区 1 的第 0 块即为绝对块号 $1 \times 4 + 0 = 4$）。




*图 B-1：MIFARE Classic 1K 16 扇区分布、数据块与扇区控制尾块 (Sector Trailer) 结构全景图*

# 附录 C：RFID / NFC 专业术语对照表

| 英文专业术语 | 中文标准译名 | 通俗含义解释 |
| --- | --- | --- |
| **RFID** | 射频识别技术 | 利用无线电高频电磁场进行非接触式双向数据识别的技术总称 |
| **NFC** | 近场通信 | RFID 的高频子集，支持卡片模拟、读卡器与点对点交互 |
| **UID** | 唯一物理标识符 | 芯片出厂时由晶圆厂激光刻录的全球唯一卡号序列号，不可篡改 |
| **ATQA** | 应答请求标识 | 卡片被 13.56 MHz 磁场激活后回送的 2 字节卡型特征字 |
| **Anti-Collision** | 防冲突算法 | 当感应场中同时出现多张卡片时，读卡器逐位仲裁锁定单张卡片的机制 |
| **Inductive Coupling** | 电感电磁耦合 | 读卡器线圈与卡片线圈通过高频交变磁通量进行无接触式电能传递 |
| **Key A / Key B** | 扇区访问密码 | MIFARE 卡每个扇区独立配置的两套 6 字节双向加密密钥 |
| **Access Bits** | 存取控制位 | 控制该扇区各块是否允许读取、写入或修改密码的 4 字节安全位 |
| **Baud Rate** | 波特率 | 串口每秒传输的二进制码元数量，常用 115200 bps |
| **XOR Checksum** | 异或校验和 | 串行通信中逐字节累加异或所得的检错冗余字节 |

# 附录 D：模块参数持久化与出厂配置映射

果蝇一号内部集成了高可靠性的非易失性参数存储机制（Flash 参数页），即使频繁断电，用户配置的所有个性化参数（波特率、工作模式、扇区密钥、防重读时间、射频增益等）均可完好保存。

## 1. 存储页结构规范

- **存储块大小**：128 字节独立参数保留页。

- **数据有效载荷**：43 字节参数字段 + 4 字节 CRC32 循环冗余校验码。
- **校验保护策略**：上电时自动校验魔数（Magic Word）、存储格式版本号（当前基线为 v15）以及全段 CRC32。只要校验通过即载入用户配置；若遇初次上电或断电误操作导致数据不完整，系统自动安全复位为出厂默认参数，绝不死机。

## 2. 参数自动平滑升级策略（v15 存储基线）

为确保不同批次与版本的软硬件无缝兼容，模块内置了向后兼容的参数迁移引擎： - 若读取到旧版有效配置（如 v14、v13 或 v11），系统自动保留原有的模块地址、波特率、工作模式、密钥与增益参数，仅将新增的扩展配置字段填充为安全的出厂默认值，免去重新配置的烦恼。

---

**手册发布单位** · 果蝇一号 DF-01 文档工程组 · 2026-10 修订版