CIU32F003 通用 Flash 存储模块使用指南

⚠️ 核心警告 (必读)

烧录工具强制要求：

本模块包含在 RAM 中运行的代码 (__RAM_FUNC) 以规避 Flash 读写冲突。

调试或量产时，必须使用官方工具 CIU32 Programmer 进行烧录。

- **原因：** 第三方工具（如 PyOCD/OpenOCD）可能无法正确处理"烧录后复位"的时序，导致程序卡死在 Flash 操作中或报 SWD 通信错误。
- **设置：** 在 `CIU32 Programmer` 中，**务必勾选** `Run after programming`（烧录后运行）选项。

---

1. 模块架构 (Architecture)

本模块采用类似 Java Spring Data 的分层设计，实现了业务逻辑与底层驱动的完全解耦。

层级	对应文件	职责 (Role)	是否需要修改
底层驱动	`drv_flash.c/.h`	**原子操作。**负责 Flash 的解锁、擦除、写入。包含 `__RAM_FUNC` 实现，解决 XIP 崩溃问题。	❌ **严禁修改**
通用框架	`lib_flash_jpa.c/.h`	**JPA 核心。**负责内存对齐、脏检查 (Dirty Check)、批量写入循环。不关心具体数据内容。	❌ **通用库，勿改**
实体层	`app_config_entity.h`	**Entity (DTO)。**定义具体的业务参数结构体（如电压、频率等）。	✅ **在此添加参数**
仓储层	`app_config_repo.c/.h`	**Repository & Service。**负责数据的加载、默认值恢复、以及 Get/Set 业务逻辑封装。	✅ **在此写业务逻辑**

---

2. 存储规格

- **存储位置：** Flash **最后一页 (Last Page)。**

- **物理地址：** `0x0000 5E00` (基于 CIU32F003 的 24KB Flash)。
- **页大小：** 512 字节。
- **有效载荷：** 除去头部 Magic 和尾部 Checksum，约 **508 字节** 可用。
- **容量估算：** 可存储约 **250 个** `uint16_t` 参数。

---

3. 快速上手：如何添加一个新参数?

假设你要添加一个名为 `contrast` (屏幕对比度) 的参数，类型为 `uint8_t`。

第一步：定义实体 (Entity)

打开 `Project_Source/Application/app_config_entity.h`，在结构体中添加字段：

C

```
typedef struct {
    uint16_t magic_head;

    // ... 原有参数 ...
    uint16_t target_voltage;

    // [NEW] 新增参数
    uint8_t  contrast;      // 0-100
    uint8_t  _padding[1];   // 手动补齐，保持结构体总长度是 2或4 的倍数(推荐)

    uint16_t checksum;
} AppConfig_Entity_t;
```

第二步：设置默认值 (Factory Reset)

打开 `Project_Source/Application/app_config_repo.c`，在 `ConfigRepo_FactoryReset` 函数中初始化它：

C

```
void ConfigRepo_FactoryReset(void) {
    // ... 其他初始化 ...
    g_repo_instance.target_voltage = 220;

    // [NEW] 设置默认对比度
    g_repo_instance.contrast = 50;

    ConfigRepo_Save();
}
```

第三步：实现 Getter/Setter (Service)

打开 `app_config_repo.c` 和 `.h`，添加访问函数：

C

```
/* In .h file */
uint8_t Service_GetContrast(void);
void    Service_SetContrast(uint8_t val);

/* In .c file */
uint8_t Service_GetContrast(void) {
    return g_repo_instance.contrast;
}

void Service_SetContrast(uint8_t val) {
    if (val > 100) return; // 业务校验
    g_repo_instance.contrast = val;
    // 注意：此处只更新 RAM，不写 Flash
}
```

第四步：在业务逻辑中调用 (Main)

C

```
/* main.c */
int main(void) {
    // 1. 初始化 (必须调用，否则数据无效)
    ConfigRepo_Init();

    // 2. 读取使用
    uint8_t con = Service_GetContrast();
    LCD_SetContrast(con);

    while(1) {
        // 3. 修改参数 (例如按键触发)
        if (Key_Pressed) {
```

```
            Service_SetContrast(60);
        }

        // 4. 保存参数 (例如退出菜单触发)
        // 底层会自动对比数据，如果没变则不会真的擦写 Flash
        if (Menu_Exit) {
            ConfigRepo_Save();
        }
    }
}
```

---

4. 关键机制说明 (FAQ)

Q1: 为什么我看不到 Flash 擦写操作?

A: 所有底层操作都被封装在 `lib_flash_jpa.c` 中。它会自动处理解锁、擦除、写入和锁定。你只需要调用 `ConfigRepo_Save()`。

Q2: 频繁调用 `Save()` 会把 Flash 写坏吗?

A: 不会。 框架内置了 脏检查 (Dirty Check) 机制。

JPA_Save 会先比较 RAM 中的数据和 Flash 中的旧数据。如果完全一致（memcmp == 0），它会直接返回 JPA_SKIPPED，完全不执行擦除和写入操作，不消耗 Flash 寿命。

Q3: 为什么结构体里要加 `padding`?

A: CIU32F003 是 **Cortex-M0+** 内核。M0+ 不支持非对齐内存访问（Unaligned Access）。虽然 JPA 框架层用了 `memcpy` 来规避这个问题，但在业务层保持数据的自然对齐（2字节或4字节）是良好的编程习惯，能提高代码执行效率。

Q4: 保存时屏幕会卡顿吗?

A: 如果数据确实发生了变化，Flash 擦写大约需要 **30ms**。

- 对于 **HT1621 段码屏**：屏幕内容会保持显示，**视觉无卡顿**。

- 对于 **串口通信**：建议在数据空闲期保存，或在协议层增加重试机制。

---

5. 目录结构参考

Plaintext

```
Project_Source/
├── Drivers/              <-- [通用框架层]
|   ├── drv_flash.c       (底层: RAM Func, 寄存器操作)
|   ├── drv_flash.h
|   ├── lib_flash_jpa.c   (中间层: JPA, 脏检查, 对齐处理)
|   └── lib_flash_jpa.h
|
└── Application/          <-- [业务逻辑层]
    ├── app_config_entity.h (定义你的参数结构)
    ├── app_config_repo.c   (参数的 Get/Set/Reset 实现)
    └── app_config_repo.h
```