GigaDevice Semiconductor Inc.

GD32 Flash 模拟 EEPROM 中间件

应用笔记

AN335

1.0 版本

(2026 年 7 月)

目录

表索引

1. 前言

本应用说明介绍了 EEMUL 的设计、架构及完整集成流程--适用于 GD32 系列微控制器的 Flash 支持的模拟 EEPROM 中间件。适用于需要在 GD32 设备上进行可靠非易失性参数存储并希望了解中间件使用方法及其工作原理的固件工程师。

本笔记按顺序进行编排。第 2 章和第 3 章建立理论基础--原始 Flash 存储器的约束以及 EEMUL 为克服这些限制所采用的设计原则。理解这些部分可使后续章节中的配置与集成决策更加直观。第 4 至第 7 章涵盖实际内容：如何配置库、如何实现硬件抽象层、如何逐步集成到项目中以及如何针对应用的耐久性需求进行设计。第 8 章的 API 参考作为完整的逐函数参考。第 9 章提供故障排查指南和常见问题。第 10 章介绍了中间件附带的演示项目。

为更好地使用本笔记，读者应熟悉 GD32 MCU 闪存控制器(FMC)的操作，并能够使用 GD32 标准外设库或裸机方式用 C 编写固件。有嵌入式非易失存储概念的基础知识更佳，但非必需--本笔记将从原理出发介绍所有必要概念。

本文档中引用的 EEMUL 源代码和模板文件由兆易创新作为 GD32 中间件包的一部分进行分发。请参考兆易创新官方软件发布获取最新版本。

注意：本应用笔记仅作参考，若与用户手册或数据手册内容有冲突，以用户手册或数据手册为准。始终根据 GD32 器件型号以用户手册核实 Flash 时序及耐久性参数。

2. Flash 存储基本原理与 EEPROM 问题

2.1. 为什么 GD32 MCU 不包含 EEPROM

真正的 EEPROM(电可擦可编程只读存储器)支持字节级擦除和编程操作，每单元耐受 10 万次及以上。芯片集成专用 EEPROM 阵列会增加芯片面积和工艺复杂度。多数现代化、成本优化的 MCU--包括 GD32 系列--仅集成片上 Flash 存储器，依靠固件完成 EEPROM 行为模拟以满足持久参数存储需求。

这并非 GD32 独有的限制。整个嵌入式行业均采用类似方法。问题在于原始 Flash 存储特性与 EEPROM 大不相同，直接用于参数存储容易导致数据损坏、硬件提前失效或两者皆有。在集成任何模拟 EEPROM 方案之前，必须了解这些差异。

2.2. 原始 Flash 的四大基本问题

Flash 存储存在四个限制，使其不能直接取代 EEPROM。每一点都是实际工程难题，EEMUL 专为解决这些问题而设计。

第一是擦除粒度问题。Flash 存储单元只能从逻辑 1(擦除状态)编程为逻辑 0(烧写状态)，不能反向。如需恢复为 1，只能擦除包含该单元的整个页。GD32 系列页大小因型号而异，容量小的几百字节，大的几万字节。这意味着仅更新参数的一个字节，也要擦除并重写整个相关页。该页上储存的其他数据需单独保留，增加固件设计复杂性，且擦除过程中存在数据丢失的风险。

第二是写入次数有限。每次擦除和重编程，Flash 存储单元都在退化。其氧化层捕获电荷能力逐步下降，达到一定循环次数后(GD32 片上 Flash 一般为 1 万次)，页面便失去可靠性。如果应用每次参数更新都写同一位置，且更新频率高，则预留 Flash 区域寿命远低于设备本身寿命。例如设备设计寿命为 10 年，每秒写一次，将产生约 3.15 亿次写操作--是单个 Flash 页面额定耐久度的 3.15 万倍。

第三是对掉电的敏感性。一次 Flash 写入操作需经历擦除(通常数毫秒)和随后编程。如果整个过程中任何时刻断电--无论在擦除后还是编程中--都可能导致数据为部分写入、不确定状态。如仅采用"擦除后写入"做法，一旦掉电无回退机制：写入时掉电后，应用无法判断数据有效性或是否损坏。

第四是不能字节级擦除。EEPROM 可对任意字节独立原位更新，而 Flash 同页内字节共享，更新一个字节需擦除重写整个页面。Flash 天生不适合细粒度、高频次单参数更新，除非增加缓冲层。

2.3. EEMUL 如何逐一解决上述问题

EEMUL 针对上述四个问题分别提供了解决机制。擦除粒度通过预留专用 Flash 区域并采用环形日志方式实现。数据不再原地更新，每个新值写入环中的新"块"，原有块仅停止作为活动块。只有整页空间用尽需要回收时才进行擦除操作。

写入耐久性通过均衡磨损(wear leveling)解决。块在整个预留区域严格顺序循环消耗，库不会在其他所有块用过之前重用同一块。这样可均匀分摊擦除次数，将有效写入寿命提高至各逻辑块数量的总和。

掉电保护通过原子提交协议实现。每次写入均按严格流程进行，新块先作为"预提交"候选写入，只有等原子方式写入最终提交标志位后才激活。如果在提交前掉电，因提交标志未写全，部分块将在下次上电自动丢弃。已提交块保持不变并继续作为活动块。

字节级更新通过基于描述符的参数模型提供。应用声明一组命名参数和其大小，EEMUL 管理包含全部参数的完整快照。对某参数的写操作将先反映到 RAM 影子缓冲区，并和当前快照比对：无字节实际变化则忽略整个写操作，无任何 Flash 活动--这个差异(无操作)忽略是主要的降低磨损机制。实际有变化时，完整快照写入新的且已擦除的块；编程过程中，EEMUL 仅跳过全部为 0xFF 的对齐单元，因为已擦除的 Flash 单元即为 0xFF。

3. EEMUL 架构

3.1. 代码库结构

EEMUL 中间件作为五个源文件发布在代码库根目录。每个文件在整体系统中职责明确。

表 3-1. 代码库文件结构

文件	用途与内容
eemul.h	完整的公共API。定义所有类型、枚举、含默认值的配置宏、eemul_handle_t运行时结构以及所有公共函数声明。应用只需包含该头文件。还包含详细Doxygen文档，涵盖全部设计。
eemul.c	完整中间件实现。包含块扫描、原子提交、均衡磨损、坏块处理、差分写检测、缓冲区管理以及所有公共API函数的内部逻辑。
eemul_port.h	硬件抽象层(HAL)接口定义。声明了需要用户实现的全局eemul_port_ops变量。同时也是MCU特定端口文件的包含点。
eemul_port_template.c	HAL端口的自说明模板实现。包含erase_page、program、read和(可选)crc32等完整的存根函数，并以注释详细说明函数的具体职责及如何用MCU厂商SDK调用替换存根。
eemul_config_template.h	模板配置头文件，列出所有可调宏、默认值及相关说明。将此文件复制到项目中重命名为eemul_config.h，再按需调整参数。

3.2. 预留 Flash 区域

EEMUL 基础是片上 Flash 的一段连续区域，该区域仅用于模拟目的，禁止存放代码或其它数据。该区域由两个编译期参数确定：EEMUL_REGION_END_ADDR(独占的终止地址)和 EEMUL_NUMBER_OF_FLASH_PAGES(物 理 Flash 页 数) 。 首 地 址 由 EEMUL_REGION_END_ADDR 减去区域总大小自动推导而得。

该区域在概念上被划分为一系列逻辑块。逻辑块是 EEMUL 读取或写入的最小单元。每个逻辑块由一个固定大小的头部和紧接其后的固定大小的数据载荷组成。头部包含用于校验和排序块的元数据；数据载荷包含按照描述符排列的所有模拟参数的完整当前值。

在链接脚本中保留该区域是一个关键步骤。如果未明确告知链接器保留此区域为空，工具链可能会在此处放置代码或数据，从而覆盖已存储的参数。GCC(LD 脚本)和 IAR(ICF 脚本)的具体链接器语法不同，详见第 6.2 节。

注意：必须至少保留两个 Flash 页(EEMUL_NUMBER_OF_FLASH_PAGES >= 2)。这一要求由 eemul.h 中的编译时#error 强制执行。原因在于 EEMUL 绝不能擦除包含当前活动块的页。如果只有一个页，则在保留活动块的同时无处写入新数据。

3.3. 块布局模式

EEMUL 支持两种物理布局模式，用于确定逻辑块如何映射到 Flash 页上。布局方式的选择将显著影响存储密度、写入耐久性和最大载荷大小。

在页(Page)模式(EEMUL_BLOCK_MODE_PAGE)下，每个逻辑块占用一个或多个完整的物理 Flash 页。块大小会向上取整到最近的页边界。这意味着 1KB 页即使只有 200 字节的数据载荷(头部+数据载荷=约 216 字节对齐后)，仍然使用整个 1KB 页。页面模式是最简单的布局方式，当载荷相对于页面大小较大，或者分析的简单性优先于存储密度时最为适用。

在子块(Sub-Block)模式(EEMUL_BLOCK_MODE_SUBBLOCKS)下，每个物理 Flash 页被分成多个固定大小的槽，每个槽是一个逻辑块。例如，如果块总大小(头部加对齐后的数据载荷)为 32 字节，Flash 页大小为 1KB，那么一页可以容纳 32 个子块。这种方式更加高效：同样 2KB 的模拟区域，在页面模式下只能容纳 2 个块，而在子块模式下可容纳 64 个子块，使写入耐久性提升 32 倍。

但代价在于子块模式要求块总大小(头部+数据载荷)能被页大小整除。如果载荷较大--例如几百字节--那么每页的子块数量会很少，这时相较于页面模式的效率提升就会变得微乎其微。对于小型参数集(总载荷小于 64 字节)，子块模式几乎总是最佳选择。

表 3-2. 块模式对比

方面	页面模式	子块模式
块大小	向页边界取整	头部 + 数据载荷，对齐到EEMUL_ALIGN_BYTES
每页块数	1(始终)	floor(页大小/块字节数)
最适用	大载荷(大于页大小一半)	小载荷；耐久性最大化
耐久性倍数	等于保留页数	每页块数 x 保留页数
擦除边界	块=页，简单	多个块共享一页；仅当整页被用尽时才擦除

3.4. 块头部

每个逻辑块都以头部开始，头部有三项主要作用：标识 Flash 区域中有效的 EEMUL 块、根据写入顺序对块排序、标记块为已提交、未提交(预提交)或永久损坏。EEMUL 提供两种头部格式，可在编译时选择。

精简头部为 16 字节，包含实现正确操作所需的最小字段。magic 字段为 32 位 ASCII 常量"EMUL"(0x4C554D45)。任何块的前四字节不匹配此常量都将被立即忽略--这可防止与已擦除 Flash(读取为 0xFFFFFFFF)或无关代码段中的随机数据混淆。sequence 字段为 32 位单调递增计数器。在区域扫描发现的所有有效块中，sequence 值最大者为最近提交的块，并成为活动块。bad_marker 字段通常为 0xFFFFFFFF，在基于 Flash 标记策略(精简头和完整头均支持)下会被写为 0xBAD0BAD0，永久退役该块。commit_flag 字段最为关键：写入开始时初始化为 0xFFFFFFFF，仅在数据载荷全部写入后，才会原子性地被清零(变为 0x00000000)。只有 magic 字段匹配、bad_marker 不是退役值且 commit_flag 为 0x00000000 时，块才被视为已提交且有效。精简头部不包含完整性 CRC--如果需要检测载荷损坏，请使用完整头部。

完整头部为 32 字节，在精简头部基础上扩展了完整性相关字段。magic 常量扩展为 64 位 ASCII 字符串"EEMULATE"(0x4554414C554D4545)。payload_len 字段记录载荷的字节数，便于检测不匹配。descriptor_crc 字段覆盖参数描述符数组，可使 EEMUL 检测固件更新后参数布局的变化。bad_marker 字段通常为 0xFFFFFFFF，当因多次硬件错误永久退役某个块时被写为 0xBAD0BAD0。完整头部还包括统一的 block_crc 字段：该字段是对前述头部字段(magic、sequence、payload_len、descriptor_crc)和整个数据载荷计算的 CRC32。由于 block_crc 一次性覆盖头部和载荷字节，任何字节位置的原位位翻转均可通过下一次读取检测出来。写入启动时，block_crc 保持在擦除值(0xFFFFFFFF)，仅在数据载荷完全写入后，在提交序列的专用 CRC 步骤中被编程(见 3.5 节)。

表 3-3. 块头部字段参考

字段	精简(16字节)	完整(32字节)	说明
magic	32位"EMUL"	64位 "EEMULATE"	头部的首个字段，先于其他字段进行校验。magic不符的块将被静默跳过。
sequence	uint32_t	uint32_t	单调递增的写计数器。所有有效块中sequence值最大的块即为活动块。
payload_len	不适用	uint32_t	期望的载荷字节数。如计算出来的载荷大小与该值不符则拒绝该块。
descriptor_crc	不适用	uint32_t	初始化时参数描述符数组的CRC。检测固件更新间描述符的变化。
block_crc	不适用	uint32_t	涵盖前20字节头部(magic、sequence、payload_len、descriptor_crc)及整个数据载荷的CRC32。在载荷写入后与descriptor_crc一起作为一个双字编程。检测头部或载荷字节的原位损坏。
bad_marker	0xBAD0BAD0=坏块	0xBAD0BAD0=坏块	两种头部格式都有(精简头偏移8，完整头偏移24)。由坏块处理程序写入，实现跨所有掉电周期永久屏蔽该块。
commit_flag	0xFFFFFFFF=预提交；0x00000000=已提交	相同	原子提交指示。只有commit_flag=0x00000000且所有其他检查通过时，块才有效。

3.5. 原子提交协议--掉电安全详解

原子提交协议是一种机制，保证无论何时断电，EEMUL 绝不会向应用程序呈现损坏或部分数据。该协议围绕 Flash 存储的物理特性设计：位仅能从 1 变为 0(编程)，而已经是 0x00000000 的字不会受影响。提交标志利用单个字的最终编程，完成块的原子性定稿。

一次完整写操作按如下八步进行。理解每一步，也可知如果在此处断电将会发生什么：

第 1 步--加载影子缓冲区：从 Flash 读取活动数据载荷至 RAM 影子缓冲区(shadow_buf)。这是一次直接的内存拷贝，将活动块数据载荷的 Flash 地址内容复制进来。此时影子缓冲区保存着所有参数的当前状态。

第 2 步--在 RAM 中应用差异：将请求的参数更改应用到影子缓冲区。EEMUL 随后将修改后的影子缓冲区与旧数据载荷逐字节比较。如果数据实际未发生变化，则立即放弃写入并返回 EEMUL_STATUS_OK，Flash 不被操作--这种差异写检测是耐久性提升的关键机制。

第 3 步--选择下一个候选块：EEMUL 选择旋转序列中的下一个块。在页面模式下，会跳过当前活动块所在的 Flash 页中的其他块(以避免擦除活动数据)；在子块模式下，多个逻辑块本就共用一页，此时只会避开活动块本身。标记为坏块的也会被跳过。如果没有安全候选块，可提交失败，eemul_write_param()/eemul_write_at()返回 EEMUL_STATUS_ERR_HW。

第 4 步--擦除候选页：如果候选块处于页面模式，或是在子块模式下为页内首个子块(代表整页已写满需回收)，则会对页面进行擦除。这是最耗时的一步(通常 1-10ms)。如果此时掉电，活动块不会受到影响；恢复后 EEMUL 会重新扫描并找到它。

步骤 5 - 写入预提交头部：只编程头部的预提交前缀--紧凑头部为 8 字节，完整头部为 16 字节--包括 magic、新序列号，在完整头部模式下还包括 payload 长度。descriptor_crc 和 block_crc 字段保持擦除值(0xFFFFFFFF)：它们稍后在步骤 7 中一起编程，等 payload 存在时，这样这个 8 字节单元只写一次(ECC Flash 的硬性要求)。commit_flag 也保持擦除值(0xFFFFFFFF)。这将块标记为"预提交候选"。如果此时掉电，块尚未有效。恢复时，EEMUL 看到该块 magic 和序列号匹配，但 commit_flag != 0x00000000，所以丢弃此块并回退到前一个活动块。

步骤 6 - 编程 payload：完整 payload 从 shadow buffer 逐字写入候选块的 Flash payload 区域。每个字以 EEMUL_ALIGN_BYTES 单位一次编程。字节全为 0xFF 的 align-unit 会跳过--因为页擦除后已经是这种状态，跳过无操作编程。这使得包含全 0xFF 内容的 payload 值(例如 uint64_t 字段等于 UINT64_MAX)在 ECC 保护的双字 Flash 上可以安全往返，否则 FMC 会拒绝全 0xFF 写入。如果此时掉电，payload 会部分写入，但 commit_flag 仍为 0xFFFFFFFF，所以该块无效。上一个活动块仍为活动块。

步骤 7 - 计算并写入 descriptor_crc 和 block_crc(仅完整头部)：EEMUL 对前 20 个头部字节(magic、sequence、payload_len、descriptor_crc)与刚编程的 payload 字节拼接后计算 CRC32。随后将 descriptor_crc 和 block_crc 作为单个 8 字节双字(DW2)编程到仍为擦除状态的单元中--一次编程，禁止重编程。如果此时掉电，双字可能只部分编程(有些位未从 1 翻转到 0)，但 commit_flag 仍为 0xFFFFFFFF，因此该块仍是预提交候选，恢复时会被丢弃。在紧凑头部模式(EEMUL_USE_FULL_BLOCK_HEADER = 0)下这一环节会完全跳过--紧凑头部不携带完整性 CRC。

步骤 8 - 原子提交：commit_flag 被编程为 0x00000000。这是一次单字 Flash 编程操作。要么操作完成，块成为新的活动块，要么先掉电操作不完成。不存在中间状态：一次写操作只会写完整个字，或者什么也不写。此步骤后，候选块通过所有有效性检查并成为新活动块。

这八个步骤保证在任意时刻掉电系统都能恢复到完全有效状态。关键在于 commit_flag 是最后写入的，并且是一次原子编程操作。在完整头部模式下，descriptor_crc 和 block_crc 会在 commit_flag 之前一起写入，因此成功提交的块总是带有已经针对实际写入的头部和 payload 字节校验过的 CRC。

3.6. 均衡擦写 - 线性轮转

EEMUL 使用线性轮转作为其均衡擦写策略。库会跟踪最近写入块的索引(active_block_index)。每次需要新写入时，候选块为 active_block_index+1，最后一个块后会回绕到 0。这样所有块在重用前被严格轮转使用。

该方法有两个重要特性。首先，它保证在区域内每个块都收到至少一次擦除周期之前，不会有块被二次擦除--最大程度均匀磨损。其次，实现起来非常简单，无需为每个块设置 RAM 中的擦写计数器，因为每个块头部中的序列号已作为上次使用的隐式时间戳。

均衡擦写的实际效果是模拟区域的写入耐久性随逻辑块数量线性提升。区域有 64 个子块则可支持单页 Flash 写入次数的 64 倍--即 10000 次耐久度可变为 64 万次应用层写入，任何页到极限之前。

3.7. 坏块处理

硬件缺陷可能导致某些 Flash 页擦除或编程失败。没有坏块处理时，单一失效页可能永久阻止新写入提交。EEMUL 提供两种可配置策略管理坏块。

重试策略(EEMUL_BADBLOCK_POLICY_RETRY)在 RAM 中为每个块维护一个错误计数器。块出现硬件失败时错误计数+1。计数到达 bad_retry_threshold，就在 RAM 中的 bad_block_flags 数组中标记并跳过本次上电剩余时间。下次上电时 RAM 标志复位，EEMUL 会再次尝试使用该块。此策略适用于存在瞬时错误(如 ESD 事件)且块可能恢复场合。

Flash 标记策略(EEMUL_BADBLOCK_POLICY_MARK)在重试策略基础上扩展：当块达到 bad_retry_threshold 错误次数时，EEMUL 还会将 bad_marker 值(0xBAD0BAD0)写入 Flash 块头部。该标记会在后续所有区域扫描中检测，永久禁止该块在所有断电周期使用。bad_marker 字段在两种头部(紧凑头偏移 8，完整头偏移 24)中均有，因此适用于两种头部模式。硬件故障预期为永久时，采用此策略。

3.8. Shadow RAM Buffer 和差分写入

每个 EEMUL 句柄在 RAM 中维护一个 shadow buffer，大小等于 payload 的总对齐大小。shadow buffer 总是包含所有参数最近提交的值。EEMUL 初始化时，通过从 Flash 读取活动块的 payload 来填充 shadow buffer。shadow buffer 作为差分写入和批量更新的暂存区(见下)，而非读取路径。读取直接来自映射 Flash 的活动块，在 GD32 设备上速度与 SRAM 一致，无 Flash 总线负担。

请求写入时，EEMUL 会从 Flash 当前活动块刷新 shadow buffer，并将请求的新值逐字节对比。如果受影响区域所有字节都未改变，写入会被完全抑制：EEMUL 直接返回 EEMUL_STATUS_OK，不消耗写入周期。此差分写入检测对于定期"保存"所有参数(无论参数是否变化)的应用尤其有价值，例如定时器触发的配置周期性保存。

成功提交后，shadow buffer 反映新存储的值。因为每次写入前都先从 Flash 刷新再对比一次，

所以 buffer 每次都与 Flash 状态重新同步。

3.9. 序列号和恢复

每个提交块携带一个 32 位递增序列号，随每次写入单调递增。扫描区域时序列号作为排序依据：若区域内发现多个有效块，则序号最大者为活动块。预提交块(commit_flag != 0x00000000)、magic 无效块以及--完整头部模式下--block_crc 与重计算 header+payload CRC 不符的块都会被自动忽略。

enable_monotonic_sequence 配置项控制断电和区域重格式化后序列号行为。为 true(推荐)时，恢复后库将 next_sequence 初始化为 active_sequence+1，确保格式化和重初始化后严格单调排序。为 false 时，格式化后序列号重置为 1，若出现序列号环绕则理论上可能导致最大与最小序号同时存在产生歧义。单调模式更安全，除非为节省追踪 next_sequence 所用的额外 RAM 时可禁用。

恢复过程(eemul_recover)会对区域内所有块线性完整扫描。它读取每个块头部，校验 magic、payload_len、descriptor_crc、bad_marker 和 commit_flag。在完整头部模式下还将读取 payload 并重新计算 header+payload CRC，对比 block_crc--校验不符则丢弃。所有幸存有效块中序号最大为胜。恢复还会从完整头部收集 bad_marker 以恢复永久坏块标志。恢复的活动块随后加载到 shadow buffer。如果没有找到有效块--如首次启动时 Flash 区域全空--EEMUL 自动格式化：区域所有页面擦除，写入第一个 payload 全 0(0x00)初始化的提交块，为所有参数提供确定性的冷启动状态。

4. 配置参考

所有编译期配置通过 eemul_config.h 定义的宏控制。库自带 eemul_config_template.h，包含了所有默认值宏。将其复制到工程 include 路径、重命名为 eemul_config.h，并根据硬件及应用需求调整值。宏由 eemul.h 在所有其他定义前处理，因此在整个库内生效。

表 4-1. 配置宏参考

宏	默认值	说明和设计指导
EEMUL_REGION_END_ADDR	0x08010000	模拟区域的专用结束地址。区域在此地址结束(但不包括该地址)。必须与Flash页边界对齐。起始地址通过 end - (pages x page_size) 推导得出。例如：将区域放置在64 KB Flash的最后2 KB，end = 0x08010000，start = 0x0800F800。
保留区域中的物理Flash页数	2	保留区域中的物理Flash页数量。库强制的最小值为2(如果指定的页数少于2，则在编译时出错)。页数越多 = 逻辑块越多 = 耐久性越高。请在耐久性需求与可用Flash预算之间权衡。
物理页大小(字节)	0x400 (1 KB)	物理页的字节大小。必须与目标器件的擦除粒度匹配。GD32的页大小从小容量器件的几百字节到大容量器件的数十KB不等；请查阅设备用户手册FMC章节获取准确值。
Flash编程对齐和粒度(字节)	4	Flash编程对齐和粒度(字节)。所有写入操作都以该单位进行。支持的值为2(半字)、4(字)和8(双字)。所选择的值必须与目标器件的FMC编程单元匹配：带ECC保护的双字Flash必须为8，而半字和字FMC可在硬件允许范围内接受较小数值。类型不匹配会导致未定义行为；端口文件应在编译时通过#error拒绝不受支持的组合。
缓冲区分配策略	0	缓冲区分配策略。0 = 所有缓冲区在编译期以静态数组方式分配(无需堆，适用于安全关键或资源受限设备)。1 = 缓冲区在eemul_init()时用malloc/calloc分配，大小按实际描述符分配。当使用多个不同描述符的EEMUL实例、或RAM有限且静态分配浪费大量内存时设为1。
最大有效载荷大小(字节)	128	字节为单位的最大支持有效载荷大小，用于静态缓冲区分配。必须>=描述符中所有参数的总和。使用静态分配(EEMUL_ENABLE_DYNAMIC_ALLOC = 0)时用于固定有效载荷和CRC缓冲区。如果

宏	默认值	说明和设计指导
		描述符有效载荷超出此限制，eemul_init()返回EEMUL_STATUS_ERR_LAYOUT。动态分配时此宏被忽略：所有缓冲区(包括完整头CRC临时缓冲区)在初始化时都按实际描述符分配。
参数偏移预计算	1	参数偏移缓存。1 = 初始化时EEMUL计算每个参数在有效载荷中的字节偏移并保存在查找表中。后续的读写操作可O(1)时间查找偏移。0 = 每次访问时通过遍历描述符重新计算偏移，节省RAM但访问时间略增。大多数应用建议启用。
最大参数数量	32	描述符中最大参数数量，仅用于EEMUL_ENABLE_PRECOMPUTE_PARAM_OFFSETS = 1时静态偏移表大小。如果描述符包含超过32个参数需增大此值。
是否使用完整块头	0	0 = 紧凑型16字节头(magic "EMUL"，序列号，bad_marker，commit_flag)。无完整性CRC，最小化头部开销，最大化每页的子块密度。1 = 完整32字节头(magic "EEMULATE"，序列号，payload_len，descriptor_crc，block_crc，bad_marker，commit_flag)。block_crc字段覆盖头部所有前述字段及整个有效载荷，提供端到端完整性保护。用于检测头部+载荷损坏、永久坏块标记和描述符变更检测。建议生产系统使用。

以下派生值由上述宏自动计算，可在应用代码中使用：

表 4-2. 派生配置值

派生宏	公式及含义
保留Flash区域总大小(字节)	EEMUL_NUMBER_OF_FLASH_PAGES x EEMUL_FLASH_PAGE_SIZE。保留Flash区域总字节数。
保留区域起始地址	EEMUL_REGION_END_ADDR - EEMUL_REGION_SIZE_BYTES。保留区域的起始地址(含)。
块头magic值	紧凑模式下为0x4C554D45("EMUL")，完整头模式下为0x4554414C554D4545("EEMULATE")。写入到每个块头。由eemul.h中的#ifndef保护，可默认配置，应用可在eemul_config.h中重写。
块头坏块标记值	0xBAD0BAD0。根据Flash标记策略，写入块头(紧凑或完整)bad_marker字段以永久性废弃此块的值。

5. 移植到 GD32 MCU

5.1. 理解 HAL 端口架构

EEMUL 设计为完全独立于任何特定 MCU 或 Flash 驱动库。通过 eemul_port_ops_t 结构体(只包含一组函数指针)与 Flash 硬件交互。该结构体在 eemul.h 中定义，并在初始化时传递给 eemul_init()。库会将该结构体指针保存在句柄中，并在擦除、编程或读 Flash 时直接调用这些函数。

中间件逻辑与硬件驱动的分离带来了实际便利：将 EEMUL 移植到新的 GD32 目标只需在新 C 源文件中实现四个函数(三个必须，一个可选)，不需要修改库源码。模板文件 eemul_port_template.c 提供了注释详细的实现框架说明每个函数的职责。

表 5-1. HAL 端口函数接口

函数	必需	签名	约定及实现说明
erase_page	是	bool (*)(uint32_t addr)	擦除addr处的一个物理Flash页。addr参数始终页对齐。擦除成功返回true，硬件报错返回false。必须阻塞直至擦除完成-不能在硬件确认完成前返回。实现时为目标器件调用相应的FMC擦除函数(页型用fmc_page_erase，扇区型用fmc_sector_erase)，并检查返回的fmc_state_enum。
program	是	bool (*)(uint32_t addr, const void *src)	将src中的EEMUL_ALIGN_BYTES字节精确编程到Flash地址addr。addr始终对齐到EEMUL_ALIGN_BYTES。成功返回true，硬件错误返回false。字编程(EEMUL_ALIGN_BYTES=4)时：将src复制到uint32_t并调用fmc_word_program()。双字编程(EEMUL_ALIGN_BYTES=8)时：调用fmc_doubleword_program()。
read	是	void (*)(uint32_t addr, void *dst, uint32_t len)	从Flash地址addr读取len字节到dst。GD32(及大多数ARM Cortex-M MCU)的内部Flash是内存映射，可由CPU直接访问。因此此函数通常实现为：memcpy(dst, (const void*)addr, len)。由于内存映射Flash不存在读取失败，不返回值。
crc32	可选	uint32_t (*)(const void *data,	计算data中len字节的CRC32校验

函数	必需	签名	约定及实现说明
		uint32_t len)	值。仅 EEMUL_USE_FULL_BLOCK_HEADER = 1时需要。如果在 eemul_port_ops_t结构体中设置为 NULL，EEMUL会自动使用内置的软件CRC32实现。如目标设备有硬件CRC单元并在此处连接可提升吞吐率。

5.2. FMC 解锁与上锁 - 关键要求

GD32 Flash 存储控制器(FMC)在执行任意擦除或编程操作前需要特定的解锁序列。解锁时需往 FMC_KEY 寄存器写入两个特定密钥(UNLOCK_KEY0 和 UNLOCK_KEY1)。操作完成后，必须通过设置 FMC_CTL 的 LK 位重新上锁。原因有二：保护 Flash 不被其他代码意外修改，以及防止失控指针 bug 破坏已存参数。

erase_page 和 program 两个 HAL 函数都必须遵循：解锁、执行操作、上锁的流程。GD32 标准外设库提供 fmc_unlock()和 fmc_lock()用于此目的。端口层中的擦除和编程函数还应在操作前调用 fmc_flag_clear()清除所有待处理的 FMC 状态标志，因为前次操作留下的标志可能导致硬件拒绝本次新操作。

注意：切勿在擦除或编程期间从被操作的 Flash 页执行代码。此时执行必须来自其他 Flash bank、SRAM 或指令缓存。单 bank 设备则必须保证 FMC 操作期间 CPU 只运行于缓存或 SRAM 中，或正确配置 FMC 等待状态。详情请查阅设备用户手册。

5.3. GD32F50x 完整接口实现

GD32F50x 系列使用 2 KB(bank0)或 4 KB(bank1)Flash 页，并支持半字或字编程粒度。以下完整端口实现基于 GD32F50x 标准外设库。实现了解锁/上锁流程、清除 bank 专属状态标志，并在每次操作后根据 FMC 状态返回正确布尔结果。

表 5-2. GD32F50x HAL 端口实现

```
/*
 * File: eemul_port_gd32f50x.c
 * Target: GD32F50x (page-based FMC, half-word/word programming)
 * Requires: gd32f50x_fmc.h from the GD32F50x Standard Peripheral Library
 */
#include "eemul_port.h"
#include "gd32f50x_fmc.h"
#include <string.h>

/* -- bank-specific flag helper ---------------------------------------- */
```

```
/*
 * GD32F50x parts larger than 512 KB are dual-bank: bank0 owns the
 * lower 512 KB and reports through FMC_STAT0, bank1 owns the upper
 * region and reports through FMC_STAT1. Every program/erase
 * evaluates the owning bank's ready state, so the bank's sticky
 * error flags must be cleared before every operation.
 *
 * fmc_flag_clear() accepts exactly one flag per call - the flag enums
 * encode a register offset plus a bit position, so OR-ing several
 * together does not form a valid combined value. Each flag of the
 * owning bank is therefore cleared with its own call.
 */
static void gd32_bank_clear_flags(uint32_t address)
{
#ifdef FMC_BANK0_END_ADDRESS
if ((FMC_BANK0_SIZE < FMC_SIZE) && (address > FMC_BANK0_END_ADDRESS)) {
fmc_flag_clear(FMC_FLAG_BANK1_END);
fmc_flag_clear(FMC_FLAG_BANK1_WPERR);
fmc_flag_clear(FMC_FLAG_BANK1_PGERR);
return;
}
#endif
fmc_flag_clear(FMC_FLAG_BANK0_END);
fmc_flag_clear(FMC_FLAG_BANK0_WPERR);
fmc_flag_clear(FMC_FLAG_BANK0_PGERR);
}

/* -- erase_page ------------------------------------------------------------ */
/*
 * Erases the Flash page that starts at the given address. The FMC
 * must be unlocked before erasing and locked after. The owning
 * bank's status flags are cleared on both sides of the call so a
 * stale flag never blocks the operation.
 */
static bool gd32_erase_page(uint32_t address)
{
fmc_unlock();
gd32_bank_clear_flags(address);

fmc_state_enum state = fmc_page_erase(address);

gd32_bank_clear_flags(address);
fmc_lock();
```

```
return (state == FMC_READY);
}

/* -- program ---------------------------------------------------------- */
/*
  * Programs exactly EEMUL_ALIGN_BYTES bytes to the given Flash
  * address. GD32F50x supports half-word (2-byte) and word (4-byte)
  * programming; the matching branch is selected at compile time.
  * src may not be alignment-aligned, so it is copied into a local
  * variable first.
  */
static bool gd32_program(uint32_t address, const void *src)
{
#if (EEMUL_ALIGN_BYTES == 4U)
/* 4-byte programming via one 32-bit word */
uint32_t word;
fmc_state_enum state;

memcpy(&word, src, 4U);
fmc_unlock();
gd32_bank_clear_flags(address);

state = fmc_word_program(address, word);

gd32_bank_clear_flags(address);
fmc_lock();
return (state == FMC_READY);
#elif (EEMUL_ALIGN_BYTES == 2U)
/* 2-byte programming via one 16-bit half-word */
uint16_t halfword;
fmc_state_enum state;

memcpy(&halfword, src, 2U);
fmc_unlock();
gd32_bank_clear_flags(address);

state = fmc_halfword_program(address, halfword);

gd32_bank_clear_flags(address);
fmc_lock();
return (state == FMC_READY);
#else
#error "EEMUL_ALIGN_BYTES must be 2 or 4 for GD32F50x"
```

```
#endif
}

/* -- read ------------------------------------------------------------------ */
/*
 * GD32F50x Flash is memory-mapped in the CPU address space starting
 * at 0x08000000. Reading is identical to reading SRAM - a simple
 * memcpy from the Flash address is all that is required.
 */
static void gd32_read(uint32_t address, void *dst, uint32_t len)
{
memcpy(dst, (const void *)address, len);
}

/* -- eemul_port_ops -------------------------------------------------------- */
/*
 * The global port operations structure required by eemul_port.h.
 * crc32 is left at NULL - EEMUL falls back to its built-in software
 * CRC32. If full-header mode is enabled and throughput matters,
 * connect the GD32F50x hardware CRC unit (configurable polynomial,
 * init value and reflection) here for a measurable speed-up.
 */
const eemul_port_ops_t eemul_port_ops = {
.erase_page = gd32_erase_page,
.program= gd32_program,
.read     = gd32_read,
/* .crc32 = NULL    (software CRC fallback) */
};
```

5.4. 关于基于扇区的 Flash 的注意事项

部分 GD32 系列采用基于扇区的 Flash 架构，擦除粒度在地址空间中变化-通常 Flash 起始有一些较小的扇区，随后是逐步增大的扇区。EEMUL_FLASH_PAGE_SIZE 必须等于持有模拟区扇区的物理擦除单元，因此预留区需完全位于同一大小的扇区内。

强烈建议将模拟区预留在单条扇区大小内。如果预留两个 16 KB 扇区，设置 EEMUL_FLASH_PAGE_SIZE = 0x4000(16 KB)。小容量负载(如总共 60 字节)，子块模式下每页 32 字节块可得 floor(16384 / 32) = 512 个子块，2 页就是 1024 个块-在任意页达到耐久极限前可支持千万级别写入。

基于扇区的产品 FMC API 通常用 fmc_sector_erase(sector_number) 替代 fmc_page_erase(address)。因此端口实现需将 EEMUL 传递的 Flash 地址换算为扇区号，可以通过查表或按已知扇区布局直接计算目标设备的扇区 ID。

注意：基于扇区的设备 EEMUL_ALIGN_BYTES 由 FMC 编程粒度决定，而非扇区布局--通常为 4(字)或 8(双字)。请查阅设备用户手册 FMC 章节确认支持值。

5.5. 关于双字编程及 ECC Flash 的注意事项

部分 GD32 芯片使用 64 位双字编程作为本地编程粒度，并常在 Code Flash 每个双字上启用硬件 ECC(纠错码)。配置 EEMUL 时需格外关注这些特性。

此类设备请将 EEMUL_ALIGN_BYTES 设置为 8。所有 EEMUL 写操作都以 8 字节单位执行，满足 FMC 硬件双字编程限制。负载大小与头部大小均 8 字节对齐。虽会稍微增大每块开销，但这是硬件正确性的要求。

ECC 在硬件层自动生效，对 EEMUL 基本透明。单比特错误读取时硬件自动纠正；双比特错误则报告为总线错误。这些是在系统层处理的，无需 EEMUL 端口层特别考虑。有两种 ECC 行为影响 EEMUL: 一是未覆盖完整双字宽度的 Flash 写入触发 ECC 错误(EEMUL 始终整 8 字节写保证无问题)；二是对已擦除全 0xFF 的双字进行编程会被 FMC 拒绝(因无法生成 ECC 奇偶)。EEMUL 对此会跳过 port program()调用，从而合法包含 0xFF 块的数据(如未初始化的字段、uint64_t 为 UINT64_MAX)可正常往返、不触发 ECC 错误。

6. 分步集成指南

本节详细介绍将 EEMUL 集成到 GD32F50x 项目的完整过程，每个步骤都结合原理讲解，而不仅仅是操作方法。示例目标为 GD32F503VG(1024 KB Flash，96 KB SRAM)，采用 Keil MDK 和 GD32F50x 标准外设库。

6.1. 步骤 1 - 添加源码文件到工程

从 EEMUL 仓库复制下列文件到项目目录。建议放在 Middleware/EEMUL/子目录：

eemul.h--公共头文件；需加入编译器头文件路径

eemul.c--中间件实现；需编译并链接

eemul_port.h--HAL 接口头文件

eemul_port_template.c--复制后重命名(如 eemul_port_gd32f50x.c)，并补充硬件实现

eemul_config_template.h--复制后重命名为 eemul_config.h，并调整参数

在 Keil MDK 中：右键工程窗口中的目标组，选择"Add Existing Files"。添加 eemul.c 和你的端口.c 文件。将 Middleware/EEMUL/目录加入编译器包含路径(Project -> Options for Target -> C/C++ -> Include Paths)。

6.2. 步骤 2 - 在链接脚本预留 Flash 区域

链接器必须设定模拟区为空，否则链接器可能会将代码、只读数据或中断向量表放入保留区域，EEMUL 运行过程中会写入覆盖。

GD32F503VG(1024 KB Flash)模拟区放在最后 8KB(bank1 的两个 4KB 页)，保留 1016KB 用于代码与只读数据 - 对大多数应用来说绰绰有余。该布局配置如下：EEMUL_REGION_END_ADDR = 0x08100000(1024 KB Flash 结束地址)，EEMUL_NUMBER_OF_FLASH_PAGES = 2，EEMUL_FLASH_PAGE_SIZE = 0x1000。

表 6-1. GD32F503VG 链接器脚本修改(GCC / LD)

```
/* GD32F503VG: 1024 KB Flash, 96 KB SRAM   */
/* Emulation region: last 8 KB of Flash (2 x 4 KB pages on bank1)*/
MEMORY
{
   FLASH   (rx)   : ORIGIN = 0x08000000, LENGTH = 1016K
   EEMUL_RGON (rx)   : ORIGIN = 0x080FE000, LENGTH = 8K
   RAM(xrw) : ORIGIN = 0x20000000, LENGTH = 96K
}

SECTIONS
```

```
{
   /* ... your standard sections (text, rodata, data, bss) ... */

   /* EEMUL region - must be kept EMPTY by the linker.   */
   /* The KEEP directive prevents the linker from discarding it. */
   .eemul_region (NOLOAD) :
   {
KEEP(*(.eemul_region))
   } > EEMUL_RGON
}
```

Keil MDK 分散文件(ARM Linker)可用 scatter 描述;IAR EWARM(.icf)用不初始化属性的 region 声明。无论哪种方式，请在编译后用链接映射文件(通常叫 project_name.map)核查预留区未与其它段重叠。

注意：修改链接脚本后请全量清理并重新编译工程。检查.map 文件确保没有代码或数据段落分布在 EEMUL_REGION_START_ADDR 和 EEMUL_REGION_END_ADDR 之间。还要核查调试器的 Flash 烧录算法能正确跳过此区域，下载时不会擦除-大多数 CMSIS-Pack 算法在链接器未使用该区时会自动跳过。

6.3. 步骤 3 - 创建 eemul_config.h

将 eemul_config_template.h 复制到项目 include 目录并改名为 eemul_config.h，编辑参数以匹配你的硬件和需求。如下示例面向 GD32F503VG，参数集较小(总负载<64 字节)，启用子块模式以获得最大耐久性，头部精简提升块密度。

表 6-2. GD32F503VG 的 eemul_config.h(子块模式，精简头部)

```
/*
  * eemul_config.h
  * EEPROM emulation configuration for GD32F503VG
  * Flash: 1024 KB, Page size: 4 KB (bank1), Word programming
  * Region: last 8 KB of Flash (0x080FE000 - 0x08100000)
  */
#ifndef EEMUL_CONFIG_H
#define EEMUL_CONFIG_H

/* Flash region definition */
#define EEMUL_REGION_END_ADDR   0x08100000U     /* exclusive end of 1024 KB Flash */
#define EEMUL_NUMBER_OF_FLASH_PAGES2U/* 2 pages = 8 KB total region*/
#define EEMUL_FLASH_PAGE_SIZE   0x1000U     /* 4 KB pages on GD32F50x bank1     */
#define EEMUL_ALIGN_BYTES   4U/* word (32-bit) programming   */

/* Buffer strategy: static (no heap required) */
#define EEMUL_ENABLE_DYNAMIC_ALLOC 0U
```

```
#define EEMUL_MAX_PAYLOAD_BYTES64U     /* must be >= sum of param sizes */

/* Parameter offset caching */
#define EEMUL_ENABLE_PRECOMPUTE_PARAM_OFFSETS    1U
#define EEMUL_MAX_PARAM_COUNT    8U     /* number of parameters in our descriptor */

/* Header format: compact (16 B) - maximizes sub-block count     */
/* Switch to 1 for full (32 B) header in production systems*/
#define EEMUL_USE_FULL_BLOCK_HEADER0U

#endif /* EEMUL_CONFIG_H */
```

使用本配置，库的布局由实际描述符负载决定--不是通过 EEMUL_MAX_PAYLOAD_BYTES，该值仅限制静态缓冲区大小。步骤 4 中定义的描述符总共 44 字节，因此得到的布局为：总区域=2 x 4096=8192 字节；块大小=头(16 字节)+按 4 字节边界对齐的负载(16+44=60 字节，已经对齐)；每页子块=floor(4096 / 60)=68；总块数=2 页 x 68 子块=136 块；写入耐久性=136 x 10,000=1,360,000 次应用写入，任何一页到达其额定耐久极限之前。

6.4. 第 4 步 - 定义参数描述符

参数描述符是 EEMUL 应用接口的核心。它是一个 uint8_t 值数组，每个元素指定一个参数的字节数。参数通过该数组中的从零开始的索引标识，这被称为参数 ID。

良好的描述符设计会将逻辑相关的参数组合在一起，并按实际应用中使用的数据类型确定大小。重要的是要使用 sizeof()而不是硬编码常量，因为编译器可以自动检测类型大小变化。如果某个参数以后扩展--例如，校准表从 16 字节扩充至 32 字节--则描述符必须更新，eemul_config.h 中的 EEMUL_MAX_PAYLOAD_BYTES 值也必须相应增加以匹配。

以下示例为电机控制应用定义了一个实际的参数集：固件版本(用于 OTA 追踪)、校准系数、错误日志、工作周期计数器和活动控制参数。

表 6-3. 参数描述符定义示例

```
/*
 * Application parameter types.
 * Using typedef arrays makes the sizes explicit and avoids magic numbers.
 */
typedef uint8_t   app_fw_version_t[4];   /* major.minor.patch.build     */
typedef uint8_t   app_calib_coeff_t[16];/* 4x float32 calibration values */
typedef uint8_t   app_error_log_t[8];     /* last 8 error codes (1 byte each)   */
typedef uint8_t   app_op_counter_t[4];   /* uint32_t operation cycle counter    */
typedef uint8_t   app_ctrl_params_t[12];/* 3x float32 active PID gains     */

/*
 * Parameter IDs - use an enum for readability and to avoid magic numbers.
 * The enum value equals the parameter index in s_descriptor[].
```

```
 */
typedef enum {
APP_PARAM_FW_VERSION   = 0,   /* 4 bytes   */
APP_PARAM_CALIB_COEFF = 1,   /* 16 bytes */
APP_PARAM_ERROR_LOG    = 2,   /* 8 bytes   */
APP_PARAM_OP_COUNTER   = 3,   /* 4 bytes   */
APP_PARAM_CTRL_PARAMS = 4,   /* 12 bytes */
APP_PARAM_COUNT /* = 5, used as array size guard */
} app_param_id_t;

/*
 * Descriptor array: one entry per parameter, value = size in bytes.
 * Total payload = 4 + 16 + 8 + 4 + 12 = 44 bytes.
 * Aligned to 4 bytes = 44 bytes (already a multiple of 4).
 */
static const uint8_t s_descriptor[APP_PARAM_COUNT] = {
[APP_PARAM_FW_VERSION]   = sizeof(app_fw_version_t),    /* 4   */
[APP_PARAM_CALIB_COEFF] = sizeof(app_calib_coeff_t),   /* 16 */
[APP_PARAM_ERROR_LOG]    = sizeof(app_error_log_t),/* 8   */
[APP_PARAM_OP_COUNTER]   = sizeof(app_op_counter_t),    /* 4   */
[APP_PARAM_CTRL_PARAMS] = sizeof(app_ctrl_params_t),   /* 12 */
};
```

6.5. 第 5 步 - 初始化中间件

EEMUL 初始化是最重要的步骤。调用 eemul_init()会顺序执行几个操作：验证输入参数和配置，分配或绑定运行时缓冲区，验证 Flash 区域几何结构，扫描模拟区域以查找最新有效的已提交块，如果未找到(首次启动)则执行自动格式化。这些子操作都可能独立失败，返回的状态码可标识出错的具体步骤。

在调用 eemul_init()之前，初始化配置结构体(eemul_init_config_t)必须完全填写。每个字段的行为影响都应理解后再设置。badblock_policy 字段决定当 Flash 块擦除或编程失败时的处理。对于量产设计，建议用 EEMUL_BADBLOCK_POLICY_MARK，因为它可以跨重启永久性避开坏块。开发期间，EEMUL_BADBLOCK_POLICY_RETRY 适合于调试时想要重试可能临时出错的块。bad_retry_threshold 字段设置一个块累计多少硬件错误后才被淘汰--2 或 3 较为合理，能够平衡对瞬时错误的容忍和及时淘汰有缺陷的块。建议始终将 enable_monotonic_sequence 设置为 true，这可防止区域循环后序列号产生歧义。

表 6-4. EEMUL 初始化与完整错误处理

```
/*
 * Module-level EEMUL handle. This struct holds all runtime state.
 * It must persist for the lifetime of the application - do not
 * declare it on the stack of a function that returns.
 */
```

```
static eemul_handle_t s_eemul;

void storage_init(void)
{
/*
 * Configuration for the emulation behavior.
 * These values are tuned for a production design:
 * - MARK policy: permanently retire pages that fail
 * - SUBBLOCKS mode: 68 sub-blocks per 4 KB page
 * - Retry threshold: retire after 2 consecutive failures
 * - Monotonic sequence: prevent wrap-around ambiguity
 */
const eemul_init_config_t cfg = {
.badblock_policy    = EEMUL_BADBLOCK_POLICY_MARK,
.block_mode     = EEMUL_BLOCK_MODE_SUBBLOCKS,
.bad_retry_threshold    = 2U,
.enable_monotonic_sequence = true,
};

eemul_status_t status = eemul_init(
&s_eemul,
&eemul_port_ops,     /* HAL functions from eemul_port_gd32f50x.c */
&cfg,
s_descriptor,
APP_PARAM_COUNT
);

switch (status) {
case EEMUL_STATUS_OK:
/* Normal case: active block found or region formatted. */
break;

case EEMUL_STATUS_ERR_PARAM:
/*
 * A configuration error: NULL pointer, invalid descriptor,
 * or malloc failure when dynamic allocation is enabled.
 * This is a firmware error - fix the configuration.
 */
Error_Handler();
break;

case EEMUL_STATUS_ERR_LAYOUT:
/*
```

```
 * The computed block size exceeds the page size, or the
 * derived region geometry is invalid. Adjust macros in
 * eemul_config.h - payload too large for page/header combo.
 */
Error_Handler();
break;

case EEMUL_STATUS_ERR_HW:
/*
 * Auto-format failed: Flash hardware returned an error
 * during erase or program. Check FMC configuration,
 * write-protection option bytes, and board power supply.
 */
Error_Handler();
break;

case EEMUL_STATUS_ERR_NOBLOCK:
/*
 * No usable blocks remain (all blocks bad).
 * This should not occur on first boot. If it does,
 * the reserved region may be write-protected.
 */
Error_Handler();
break;
}
}
```

6.6. 步骤 6 - 读取参数

通过 eemul_read_param()读取参数会将参数值从内存映射 Flash 中的活动块复制到应用程序提供的目标缓冲区。在 GD32 设备上，Flash 映射到 CPU 地址空间，因此读取只是一个简单的内存复制操作，在几个周期内完成，无总线冲突，无阻塞，无擦除/编程交互。活动块始终保存最新提交的值。

目标缓冲区的大小必须与描述符中声明的参数大小完全一致。EEMUL 在运行时验证此匹配，若不一致则返回 EEMUL_STATUS_ERR_PARAM。对描述符条目和目标缓冲区使用相同的 typedef(如描述符示例所示)可确保大小自动同步。

表 6-5. 读取参数 - 示例

```
/* -- Reading a single parameter ------------------------------ */

app_fw_version_t fw_ver;
eemul_status_t st = eemul_read_param(&s_eemul, APP_PARAM_FW_VERSION, &fw_ver);
if (st == EEMUL_STATUS_OK) {
```

```
/* fw_ver[0] = major, fw_ver[1] = minor, etc. */
printf("FW: %d.%d.%d.%d\n", fw_ver[0], fw_ver[1], fw_ver[2], fw_ver[3]);
}

/* -- Reading calibration coefficients ------------------------ */

app_calib_coeff_t calib;
eemul_read_param(&s_eemul, APP_PARAM_CALIB_COEFF, &calib);

/* Interpret the bytes as four float32 values */
float offset, gain, temp_comp, reserved;
memcpy(&offset,&calib[0],   sizeof(float));
memcpy(&gain,   &calib[4],   sizeof(float));
memcpy(&temp_comp, &calib[8],   sizeof(float));

/* -- Raw offset read (advanced use) ---------------------------- */

/*
 * eemul_read_at() reads directly from the active block payload in Flash by byte offset.
 * Useful when accessing sub-fields of a parameter or reading partial data.
 * The offset is the byte position within the full payload image.
 */
uint16_t param_offset = eemul_get_param_offset(&s_eemul, APP_PARAM_ERROR_LOG);
uint8_t   first_error;
eemul_read_at(&s_eemul, param_offset, &first_error, sizeof(first_error));
```

6.7. 步骤 7 - 写入参数(单个和原子操作)

使用 eemul_write_param()写入单个参数会触发 3.5 节中描述的完整八步原子提交过程。此调用被阻塞直到新的块已成功提交到 Flash 或发生错误。返回后，如果写入成功，则影子缓冲区已更新以反映新值。

差分写检测意味着，使用与当前存储值相同的数据调用 eemul_write_param()是安全且高效的--库检测到匹配，立即返回 EEMUL_STATUS_OK 且不会有任何 Flash 操作。因此，应用程序可以在定期保存过程中无条件调用 eemul_write_param()，而不必担心 Flash 过早损耗。

写入的错误处理很重要。EEMUL_STATUS_ERR_HW 表示擦除或编程期间发生 Flash 硬件故障。在这种情况下，EEMUL 已尝试写入，块策略已更新坏块状态。应用程序通常应记录错误并继续运行--先前的活动块仍然有效。当没有可用的安全候选块(所有剩余块都是坏的或不安全)时，也会返回同样的 EEMUL_STATUS_ERR_HW 代码，这表明模拟区域已耗尽或严重退化。写入 API 不会返回 EEMUL_STATUS_ERR_NOBLOCK；该代码只会出现在 eemul_init() / eemul_recover()格式化路径中。

表 6-6. 写入参数 - 示例与错误处理

```
/* -- Writing the firmware version at startup ------------------- */

const app_fw_version_t new_fw = { 1, 2, 0, 7 };     /* v1.2.0 build 7 */

eemul_status_t st = eemul_write_param(&s_eemul, APP_PARAM_FW_VERSION, &new_fw);
if (st != EEMUL_STATUS_OK) {
/* Log the error - do not halt; old data remains valid */
log_error(LOG_STORAGE, st);
}

/* -- Incrementing the operation counter ------------------------ */

app_op_counter_t counter;
eemul_read_param(&s_eemul, APP_PARAM_OP_COUNTER, &counter);

uint32_t count;
memcpy(&count, counter, sizeof(uint32_t));
count++;
memcpy(counter, &count, sizeof(uint32_t));

eemul_write_param(&s_eemul, APP_PARAM_OP_COUNTER, counter);
/* If count was unchanged (overflow to same value), no Flash write occurs */

/* -- Appending to the error log ------------------------------- */

app_error_log_t log;
eemul_read_param(&s_eemul, APP_PARAM_ERROR_LOG, &log);

/* Shift existing entries and add new error at index 0 */
memmove(&log[1], &log[0], 7);
log[0] = (uint8_t)new_error_code;

eemul_write_param(&s_eemul, APP_PARAM_ERROR_LOG, log);
```

6.8. 步骤 8 - 批量更新(多个参数，一次提交)

批量更新允许在单个 Flash 写入周期内对多个参数进行原子更改。这在两个或多个参数逻辑关联时至关重要--例如校准系数及其关联校验和，或一组所有值必须相互一致的控制参数。如果为每个参数单独调用一次 eemul_write_param()，则如果在调用之间电源中断，系统会处于部分更新状态，有些参数为旧值，有些为新值。

批量更新通过在影子缓冲区中暂存所有更改且不提交，然后将整个影子缓冲区作为一个新块提

交。流程如下：调用 eemul_begin_batch()将活动块已提交的有效载荷加载到影子缓冲区，建立已知基线；对每个要更改的参数调用 eemul_update_param_in_shadow()；调用 eemul_commit_batch() 将完整更新的影子缓冲区写入 Flash 作为新块。如果 eemul_commit_batch()失败，则不会提交任何新值--先前的块仍为活动块。

批量更新还提供性能优势：n 个参数更改只需一次写入周期，而非 n 个单独的 Flash 写入周期。在每个写入周期耗时数毫秒的 Flash 硬件上，这能显著减少 CPU 受阻于 Flash 操作的时间。

表 6-7. 批量更新 - 原子多参数提交

```
/*
 * Scenario: updating calibration coefficients AND PID gains together.
 * Both must be stored atomically - an inconsistent state (new gains
 * with old calibration, or vice versa) would cause incorrect behavior.
 */

/* New values computed by the calibration routine */
app_calib_coeff_t new_calib;
app_ctrl_params_t new_ctrl;

/* ... (populate new_calib and new_ctrl from calibration algorithm) ... */

/* Begin batch: snapshot current shadow buffer */
eemul_status_t st = eemul_begin_batch(&s_eemul);
if (st != EEMUL_STATUS_OK) {
log_error(LOG_STORAGE, st);
return;
}

/* Stage the calibration coefficients in the shadow buffer (no Flash write) */
st = eemul_update_param_in_shadow(&s_eemul,
    APP_PARAM_CALIB_COEFF,
    &new_calib,
    sizeof(new_calib));
if (st != EEMUL_STATUS_OK) {
log_error(LOG_STORAGE, st);
return;   /* batch is abandoned; no Flash changes made */
}

/* Stage the PID gain parameters (no Flash write) */
st = eemul_update_param_in_shadow(&s_eemul,
    APP_PARAM_CTRL_PARAMS,
    &new_ctrl,
    sizeof(new_ctrl));
if (st != EEMUL_STATUS_OK) {
```

```
log_error(LOG_STORAGE, st);
return;
}

/*
 * Commit: write the entire updated shadow buffer as a new Flash block.
 * After this returns EEMUL_STATUS_OK, both parameters are atomically
 * committed. If power fails before commit completes, neither update
 * is visible - the old values remain the active block.
 */
st = eemul_commit_batch(&s_eemul);
if (st != EEMUL_STATUS_OK) {
log_error(LOG_STORAGE, st);
/* old calibration and PID values remain in effect - safe to continue */
}
```

eemul_commit_batch()成功返回后，新写块即为活动块。随后通过 eemul_read_param()读取会立即返回新值。

7. 设计决策与实用建议

7.1. 选择正确的块模式

页面模式和子块模式的选择是最重要的配置决策。决策规则很简单：如果总载荷大小(头部+对齐载荷)大于页面大小的一半，则使用页面模式。否则使用子块模式。因为只有当一个页面能容纳多个子块时，子块模式才有密度优势。如果载荷太大只能每页放一个子块，则子块模式无优势，反而有更多管理开销。

对于绝大多数模拟 EEPROM 用例--存储设备配置、校准数据、计数器、错误日志--载荷远小于 200 字节，页面大小 1KB 或更大。这种情况下几乎总是应该选择子块模式。

7.2. 根据耐久性需求确定模拟区域大小

要确定模拟区域大小，请从耐久性需求反推。首先估算写入速率：应用每小时(或每天)调用 eemul_write_param()或 eemul_commit_batch()多少次？再乘以预期产品寿命(小时)。这样得到所需的应用级写入次数。

接着，计算建议配置下的逻辑块数量：子块模式下，total_blocks=(页数 x 页面大小)/块大小；页面模式下，total_blocks=页数。每个 Flash 页面可承受大约 10,000 次擦除周期，因此总 Flash 级寿命=total_blocks x 10,000。如果结果大于应用级写入需求，则配置是足够的。

表 7-1. 寿命计算示例

情景	计算和结果
工业传感器，每分钟写入一次，使用寿命10年	所需写入次数：60 x 24 x 365 x 10 = 5,256,000。子块模式，1 KB页，48字节块：21个子块/页 x 4页 = 84块。耐久度 = 84 x 10,000 = 840,000次写入。不足。增加到8页：168块 x 10,000 = 1,680,000。仍然不足。解决方案：使用两个16 KB扇区(EEMUL_FLASH_PAGE_SIZE = 0x4000)。使用48字节块：floor(16384 / 48) = 341个子块/页 x 2页 = 682块。耐久度 = 682 x 10,000 = 6,820,000次写入。1.3倍裕度，足够。
消费设备，每小时写入一次，5年寿命	所需写入次数：24 x 365 x 5 = 43,800。子块模式，1 KB页，80字节块：12个子块/页 x 2页 = 24块。耐久度 = 24 x 10,000 = 240,000次写入。裕度大(5.5倍)，足够。
安全设备，每次上电写入一次，20年寿命，100,000次上电周期	所需写入次数：100,000。子块模式，1 KB页，48字节块：21个子块/页 x 2页 = 42块。耐久度 = 42 x 10,000 = 420,000次写入。4.2倍裕度，足够。

7.3. 完整头与紧凑头对比

在产品设计中，除非受 ROM 或块密度限制不切实际，否则应使用完整头。完整头提供：有效载荷长度校验(捕捉乱序长度字段的块)和描述符 CRC 校验(检测固件更新时未擦除区域而更改

参数布局)。这些校验会增加初始化时间 3-4 毫秒(因 Flash 额外读取)，但运行时无额外开销。(注意，闪存标记坏块政策在任一头格式下均可用，见第 3.7 节。)

紧凑头适用于早期开发、参数布局仍在频繁变动的原型阶段，或子块密度最大化对空间极度敏感的环境。在固件量产前应始终切换至完整头。

注意：仅当使用 EEMUL_BADBLOCK_POLICY_MARK(在任一头格式下)时，bad_marker 字段才会写入 Flash。采用 EEMUL_BADBLOCK_POLICY_RETRY 时，bad_marker 字段只出现在头结构中，不会被坏块处理器写入。当 EEMUL_USE_FULL_BLOCK_HEADER = 1 时，完整头在此情况下仍可用于 payload_len 和 descriptor_crc 的校验。

7.4. 默认参数值初始化

当 EEMUL 首次格式化空白区域(首次开机自动格式化)时，初始块的载荷将全部置零(0x00)，以获得可预测的冷启动状态。这意味着第一次开机时，eemul_read_param()会返回填充为 0x00 的缓冲区，除非应用程序显式写入默认值。

正确做法是在首次开机 eemul_init()之后立即写入默认值。检测首次开机的最简单办法是检查一个正常情况下绝不会为零的特定参数--比如主固件版本号字节总是 1 或大于 1。若主版本号读取为 0x00，则表明区域刚刚被格式化，应写入默认值。

表 7-2. 首次启动默认值初始化模式

```
void storage_init_with_defaults(void)
{
/* ... (eemul_init as shown previously) ... */

/* Check if this is first boot by reading a sentinel parameter */
app_fw_version_t fw_ver;
eemul_read_param(&s_eemul, APP_PARAM_FW_VERSION, &fw_ver);

if (fw_ver[0] == 0x00) {
/* First boot detected (zero-initialized): write factory defaults */
const app_fw_version_t    default_fw    = { 1, 0, 0, 0 };
const app_calib_coeff_t default_calib     = { /* all zeros */ };
const app_error_log_t     default_log = { 0xFF, 0xFF, 0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF };
const app_op_counter_t    default_counter = { 0, 0, 0, 0 };
const app_ctrl_params_t default_ctrl= { /* factory PID gains */ };

/* Use a batch write to commit all defaults in one Flash write cycle */
eemul_begin_batch(&s_eemul);
eemul_update_param_in_shadow(&s_eemul, APP_PARAM_FW_VERSION,
  &default_fw, sizeof(default_fw));
eemul_update_param_in_shadow(&s_eemul, APP_PARAM_CALIB_COEFF,
  &default_calib, sizeof(default_calib));
```

```
eemul_update_param_in_shadow(&s_eemul, APP_PARAM_ERROR_LOG,
  &default_log, sizeof(default_log));
eemul_update_param_in_shadow(&s_eemul, APP_PARAM_OP_COUNTER,
  &default_counter, sizeof(default_counter));
eemul_update_param_in_shadow(&s_eemul, APP_PARAM_CTRL_PARAMS,
  &default_ctrl, sizeof(default_ctrl));
eemul_commit_batch(&s_eemul);
}
}
```

8. API 参考

8.1. 状态码

所有 EEMUL API 函数均返回 eemul_status_t 值，指示成功或具体的失败类型。应用程序应始终检查每次 API 调用的返回值，尤其是写操作，因为可能由于瞬态或永久性 Flash 硬件错误失败。

表 8-1. eemul_status_t 返回码

代码	值	含义及建议响应
EEMUL_STATUS_OK	0	操作成功完成。对于写操作，数据已提交到Flash，或因数据未变(差异检测)而跳过写入。
EEMUL_STATUS_ERR_PARAM	1	无效参数或配置错误。原因包括：句柄或指针为NULL、参数ID超出范围、大小参数与描述符不匹配、启用动态分配时malloc失败。此错误表明固件存在缺陷，开发阶段应视为致命错误。
EEMUL_STATUS_ERR_HW	2	Flash硬件操作失败。erase_page或program HAL函数返回false。失败块的坏块计数已增加。前一活动块仍然有效。记录错误并继续--除非系统需要写入100%成功，否则不应视为致命错误。
EEMUL_STATUS_ERR_LAYOUT	3	无效Flash布局。由配置宏和描述符推导的块大小超出页大小，有效载荷超出EEMUL_MAX_PAYLOAD_BYTES，或检测到其他几何不一致。请检查EEMUL_FLASH_PAGE_SIZE、EEMUL_ALIGN_BYTES、EEMUL_MAX_PAYLOAD_BYTES和描述符总载荷大小。始终属于配置错误--初始化时配置正确则运行时不会发生。
EEMUL_STATUS_ERR_NOBLOCK	4	未找到可用块。当恢复时无已提交块且自动格式化失败时，eemul_init()/eemul_recover()会返回本错误。运行时写入和批量提交失败--包括无可用候选块时--将报告为EEMUL_STATUS_ERR_HW，而非此代码。此为严重情况，表示模拟区域完全耗尽或发生大面积Flash故障。应增加模拟区域或视为系统级故障处理。

8.2. 初始化及生命周期函数

表 8-2. 初始化及生命周期 API

函数	返回值	说明
eemul_init(handle, port_ops, init_config, param_descriptor, param_count)	eemul_status_t	初始化句柄。校验所有输入，绑定HAL，预计算布局和偏移量，扫描

函数	返回值	说明
		Flash查找活动块，如果未找到有效块则自动格式化。必须在其他API调用前调用。本句柄在EEMUL会话期间需保持有效(不能被分配在会返回的函数栈上)。
eemul_deinit(handle)	void	反初始化句柄。如果EEMUL_ENABLE_DYNAMIC_ALLOC = 1，则释放所有动态分配的缓冲区。调用后需重新初始化句柄。静态分配模式下，此函数相当于无操作，但会清除内部状态。
eemul_recover(handle)	eemul_status_t	重新扫描Flash并恢复活动块。适用于外部Flash操作(如固件OTA升级)可能改变模拟区域内容后。初始化期间内部自动调用。重新加载完整头中的坏块标记并恢复序列跟踪。

8.3. 读写函数

表 8-3. 读写 API

函数	返回值	说明
eemul_read_param(handle, param_id, dst)	eemul_status_t	将活动块中param_id参数的当前值从内存映射Flash复制到dst。dst必须指向和描述符中param_id声明大小完全一样的缓冲区。在GD32上，该读取为直接内存复制，无总线损耗。如果param_id超出范围或dst为NULL，则返回EEMUL_STATUS_ERR_PARAM。
eemul_write_param(handle, param_id, src)	eemul_status_t	原子提交param_id参数的新值。采用差异写检测：若新值和当前shadow缓冲区值相同，则直接返回EEMUL_STATUS_OK，不写入Flash。否则执行完整八步原子提交。Flash出错或无候选块时返回EEMUL_STATUS_ERR_HW。
eemul_read_at(handle, offset, dst, len)	eemul_status_t	从活动块有效载荷(内存映射Flash)指定的offset字节开始读取len字节。offset为有效载荷中的绝对字

函数	返回值	说明
		节位置，并非参数ID。可用eemul_get_param_offset()获取特定参数偏移。用于读取分字段时无需了解参数完整结构。
eemul_write_at(handle, offset, src, len)	eemul_status_t	对有效载荷offset处写入len字节(原子提交)。同样适用差异写检测。适用于更新大参数的子字段而无需整体重写。

8.4. 批量更新函数

表 8-4. 批量更新 API

函数	返回值	说明
eemul_begin_batch(handle)	eemul_status_t	为多参数批量更新做准备。将活动块已提交载荷从Flash加载到shadow buffer，建立已知基线。本函数调用后，shadow buffer可接收多次eemul_update_param_in_shadow()参数更新，最后一次性eemul_commit_batch()。句柄为NULL或无效时返回EEMUL_STATUS_ERR_PARAM。
eemul_update_param_in_shadow(handle, param_id, src, len)	eemul_status_t	在影子缓冲区中暂存参数更新但不提交到Flash。len必须等于描述符中param_id声明的大小。在eemul_begin_batch()和eemul_commit_batch()之间可多次调用。大小不匹配或ID无效时返回EEMUL_STATUS_ERR_PARAM。
eemul_commit_batch(handle)	eemul_status_t	将所有已暂存的影子缓冲区更改作为单个原子块提交到Flash。如果在此过程中断电，所有暂存更改都不会被提交--先前的活动块仍然有效。提交成功后，影子缓冲区反映所有暂存更改。Flash失败或无可用候选块时返回EEMUL_STATUS_ERR_HW。

8.5. 实用函数

表 8-5. 实用 API

函数	返回值	说明
eemul_get_payload_size(handle)	uint16_t	返回从初始化时描述符中计算出的总对齐有效载荷大小(字节)。用于日志、诊断和验证EEMUL_MAX_PAYLOAD_BYTES的大小是否正确。
eemul_get_param_offset(handle, param_id)	uint16_t	返回payload镜像中参数param_id的字节偏移量。启用预计算偏移时，这是一次O(1)表查找。param_id无效时返回0。
eemul_get_param_size(handle, param_id)	uint16_t	返回描述符中指定的参数param_id的声明大小(字节)。超出范围时返回0。

9. 故障排查

本节描述使用 EEMUL 时遇到的最常见集成与运行时问题、根本原因及其解决方法。

表 9-1. 故障排查指南

现象	根本原因	解决方法
eemul_init()返回 EEMUL_STATUS_ERR_LAYOUT	计算出的块大小(头+对齐后的有效载荷)大于 EEMUL_FLASH_PAGE_SIZE，或有效载荷超出了 EEMUL_MAX_PAYLOAD_BYTES。在子块模式下，块超大是致命错误；在页模式下，只要块适合于多页块边界中就是有效的。	要么减少参数总有效载荷以便和头一起放入一页，要么增大 EEMUL_MAX_PAYLOAD_BYTES以覆盖描述符，或者从紧凑头切换到完整头以了解确切大小，或者切换到页模式并增加 EEMUL_NUMBER_OF_FLASH_PAGES为每个块提供更多空间。
eemul_init()返回 EEMUL_STATUS_ERR_HW	自动格式化失败：在首次启动区域初始化期间擦除或编程返回错误。常见原因：EEMUL_REGION_END_ADDR或PAGE_SIZE设置错误(与实际硬件不符)，保留区域启用了Flash写保护选项字节，或FMC电压或时序约束不满足。	请根据设备数据手册验证 EEMUL_REGION_END_ADDR和 EEMUL_FLASH_PAGE_SIZE。读取选项字节检查扇区保护(SPC/WP字段)。验证FMC配置，包括等待状态、供电电压，并确保正确调用fmc_unlock()。
首次启动后参数始终读回0x00	首次启动时Flash为空。EEMUL自动格式化区域并将首个有效载荷初始化为全零(0x00)。这是预期行为--应用必须在初始化后写入默认值。	按照7.4节所示实现首次启动检测模式。检测到首次启动后，使用 eemul_begin_batch/update/commit_batch在一次Flash写入中写入所有出厂默认值。
eemul_write_param()成功但断电后数据丢失	该写操作被检测为无效操作(差异检测)。影子缓冲区内容与src相同，所以没有发生Flash写入--但影子缓冲区在断电后不会持久化。Flash区域可能为空或包含旧值。	启动时写入前务必先读取参数。如果读取结果为新格式化的值(0x00)，写入默认值。利用调试功能，通过内存查看器读取EEMUL_REGION_START_ADDR处的数据块，确保真正写入Flash的字节正确。
eemul_write_param()返回 EEMUL_STATUS_ERR_HW且Flash无明显故障	无可用安全候选块：区域内所有逻辑块要么损坏，要么属于当前活动页(安全轮换会阻止擦除活动页)，	增加 EEMUL_NUMBER_OF_FLASH_PAGES或切换到子块模式以增加逻辑块数量。检查是否有块被错误退役(可能因测

现象	根本原因	解决方法
	因此提交失败并报告EEMUL_STATUS_ERR_HW。如果区域很小或所有块已永久退役，则会发生该情况。(写入API绝不会返回EEMUL_STATUS_ERR_NOBLOCK。)	试期间发生硬件故障)。注意bad_retry_threshold=0会被内部自动改为1，因此最小有效阈值为1--设置为0不会导致立即退役。
eemul_write_param()返回EEMUL_STATUS_ERR_PARAM	最常见的原因是eemul_update_param_in_shadow()的len参数与描述符中该参数ID声明的大小不一致。次常见原因：param_id超出param_count，或handle/src指针为NULL。	请确保调用处使用的sizeof()与描述符中相同类型的sizeof()一致。描述符条目和写入缓冲区都使用相同的typedef(如app_calib_coeff_t)可避免此类错误。开发期间启用断言以捕获不匹配。
固件升级后更改描述符导致数据损坏	Flash中存储的有效载荷布局由旧固件描述符创建。新固件具有不同的参数数量或顺序，导致字节偏移错误。将旧数据按新区读取会导致数据混乱。	将描述符版本字段作为第一个参数(ID 0)实现。固件升级后，读取ID 0并与期望版本比较。如不同，则写入整套默认值或迁移旧数据到新布局。设置EEMUL_USE_FULL_BLOCK_HEADER=1以启用描述符CRC检测。

10. 演示项目

已准备好构建的演示项目存放于 demos/目录，适用于多款 GD32 开发板。每块板包含两套互补的演示，均配有 main.c 源码、针对开发板定制的 eemul_config.h、MCU 专用移植文件及通用工具链(Keil、IAR、Eclipse/GD32EBuilder)工程文件。两个演示都通过 EVAL_COM 以 115200-8-N-1 波特率输出，可在串口终端(与板载 USB 转 UART 口连接)上观察进度。

10.1. EEMUL_Simple_Demo

简单演示展示了日常应用场景：一个可在每次断电后保留的小型参数存储区(启动计数器、模式、阈值等)。首次启动时通过批量 API 写入出厂默认值；后续每次启动时读取当前值、打印、递增启动计数器并保存新值。该演示采用紧凑头格式，不做错误注入，推荐作为新应用移植 EEMUL 的起点。

10.2. EEMUL_Functional_Demo

功能演示在一系列自检场景下验证 EEMUL：首启默认、重启持久化、差分写抑制、磨损均衡轮换、中断写恢复，以及(在完整头模式下)基于块 CRC 的有效载荷篡改检测。每个场景分步骤打印运行结果并标记通过/失败，结尾总结通过场景数。有些场景通过端口绑定操作主动将畸形块伪造到空闲 Flash 槽中复现中断写与数据篡改，然后确认 EEMUL 能够安全恢复。该演示适用于 EEMUL 在新 MCU 移植时的硬件上电验证。

11. 版本历史

表 11-1. 版本历史

版本号.	说明	日期
1.0	首次发布	2026 年 07 月 13 日

Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the "Product"), is owned by the Company according to the laws of the People's Republic of China and other applicable laws. The Company reserves all rights under such laws and no Intellectual Property Rights are transferred (either wholly or partially) or licensed by the Company (either expressly or impliedly) herein. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

To the maximum extent permitted by applicable law, the Company makes no representations or warranties of any kind, express or implied, with regard to the merchantability and the fitness for a particular purpose of the Product, nor does the Company assume any liability arising out of the application or use of any Product. Any information provided in this document is provided only for reference purposes. It is the sole responsibility of the user of this document to determine whether the Product is suitable and fit for its applications and products planned, and properly design, program, and test the functionality and safety of its applications and products planned using the Product. The Product is designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only, and the Product is not designed or intended for use in (i) safety critical applications such as weapons systems, nuclear facilities, atomic energy controller, combustion controller, aeronautic or aerospace applications, traffic signal instruments, pollution control or hazardous substance management; (ii) life-support systems, other medical equipment or systems (including life support equipment and surgical implants); (iii) automotive applications or environments, including but not limited to applications for active and passive safety of automobiles (regardless of front market or aftermarket), for example, EPS, braking, ADAS (camera/fusion), EMS, TCU, BMS, BSG, TPMS, Airbag, Suspension, DMS, ICMS, Domain, ESC, DCDC, e-clutch, advanced-lighting, etc.. Automobile herein means a vehicle propelled by a self-contained motor, engine or the like, such as, without limitation, cars, trucks, motorcycles, electric cars, and other transportation devices; and/or (iv) other uses where the failure of the device or the Product can reasonably be expected to result in personal injury, death, or severe property or environmental damage (collectively "Unintended Uses"). Customers shall take any and all actions to ensure the Product meets the applicable laws and regulations. The Company is not liable for, in whole or in part, and customers shall hereby release the Company as well as its suppliers and/or distributors from, any claim, damage, or other liability arising from or related to all Unintended Uses of the Product. Customers shall indemnify and hold the Company, and its officers, employees, subsidiaries, affiliates as well as its suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Product.

Information in this document is provided solely in connection with the Product. The Company reserves the right to make changes, corrections, modifications or improvements to this document and the Product described herein at any time without notice. The Company shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Information in this document supersedes and replaces information previously supplied in any prior versions of this document.