GigaDevice Semiconductor Inc.

GD32 Flash EEPROM Emulation Middleware

Application Note
AN335

Revision 1.0

(July 2026)

Table of Contents

List of Tables

1. Foreword

This Application Note describes the design, architecture, and complete integration procedure of EEMUL - a Flash-backed EEPROM emulation middleware for GD32 series microcontrollers. It is intended for firmware engineers who need reliable, non-volatile parameter storage on GD32 devices and wish to understand not only how to use the middleware, but also why it works the way it does.

The document is organized to be read in order. Sections 2 and 3 establish the theoretical foundation - the constraints of raw Flash memory and the design principles EEMUL uses to overcome them. Understanding these sections will make the configuration and integration decisions in later sections much more intuitive. Sections 4 through 7 cover the practical aspects: how to configure the library, how to implement the hardware abstraction layer, how to integrate it into a project step by step, and how to design for the application's endurance requirements. The API reference in Section 8 serves as a complete function-by-function reference. Section 9 provides troubleshooting guidance and common pitfalls. Section 10 walks through the demo projects shipped with the middleware.

To make best use of this document, the reader should have a working knowledge of GD32 MCU Flash Memory Controller (FMC) operation and be comfortable writing firmware in C using either the GD32 Standard Peripheral Library or a bare-metal approach. Background knowledge of embedded non-volatile storage concepts is helpful but not required - this document introduces all necessary concepts from first principles.

The EEMUL source code and template files referenced throughout this document are distributed by GigaDevice as part of the GD32 middleware package. Refer to the official GigaDevice software release for the latest version.

Note: This application note is for reference only. In case of any conflict with the device user manual or datasheet, the user manual or datasheet, whichever applies, shall prevail. Always verify Flash timing and endurance specifications against the GD32 device user manual for your specific part number.

2. Flash Memory Fundamentals and the EEPROM Problem

2.1. Why GD32 MCUs Do Not Have EEPROM

True EEPROM (Electrically Erasable Programmable Read-Only Memory) supports byte-level erase and program operations with endurance ratings of 100,000 cycles or more per cell. Including a dedicated EEPROM array on-chip adds die area and process complexity. Most modern cost-optimized MCUs - including the GD32 family - instead provide only on-chip Flash memory and rely on firmware to emulate EEPROM behavior where persistent parameter storage is required.

This is not a limitation unique to GD32. The same approach is used across the embedded industry. The challenge is that raw Flash memory behaves very differently from EEPROM, and using it naively for parameter storage leads to data corruption, premature hardware failure, or both. Understanding these differences is essential before integrating any EEPROM emulation layer.

2.2. The Four Fundamental Problems with Raw Flash

Flash memory imposes four constraints that make it unsuitable as a drop-in replacement for EEPROM. Each of these is a genuine engineering problem that EEMUL is specifically designed to solve.

The first problem is erase granularity. A Flash memory cell can only be programmed from a logical 1 (erased state) to a logical 0 (programmed state) - never the reverse. To restore a cell to 1, the entire page containing it must be erased as a unit. Page sizes vary across the GD32 line, from a few hundred bytes on small-capacity parts to tens of kilobytes on parts with larger Flash. This means that updating even a single byte of a parameter requires erasing and rewriting the entire surrounding page. Any other data stored on that page must be preserved separately, which complicates firmware design and introduces risk of data loss during the erase operation.

The second problem is limited write endurance. Flash memory cells degrade each time they are erased and reprogrammed. The oxide layer that traps charge gradually deteriorates, and after a finite number of cycles - typically 10,000 for GD32 internal Flash - a page becomes unreliable. If an application writes to the same Flash location on every parameter update, and those updates happen frequently, the reserved Flash region will fail orders of magnitude sooner than the device lifetime. A device expected to operate for 10 years at one parameter write per second will attempt approximately 315 million writes - 31,500 times the rated endurance of a single Flash page.

The third problem is power-loss sensitivity. A Flash write cycle takes time: the page must be

erased (typically several milliseconds), then the new data must be programmed. If power is removed at any point during this sequence - between erasing and completing the program, or in the middle of programming - the data can be left in a partially-written, indeterminate state. A simple approach of "erase then write" has no recovery mechanism: after a power failure during the write, the application cannot know whether the data is valid or corrupt.

The fourth problem is the absence of byte-level erase. With EEPROM, any byte can be independently updated in place. With Flash, the byte shares a page with many other bytes, and updating it means erasing and rewriting the entire page. This makes Flash inherently unsuitable for fine-grained, high-frequency updates to individual parameters without a buffering layer.

2.3. How EEMUL Solves Each Problem

EEMUL addresses each of these four problems with a specific mechanism. Erase granularity is handled by reserving a dedicated Flash region and treating it as a circular log. Instead of updating data in place, each new value is written to a fresh location - a new "block" in the ring. The old block is left in place and simply ceases to be the active block. Erase operations only occur when an entire page has been consumed and needs to be recycled for future use.

Write endurance is addressed by wear leveling. Blocks are consumed in strict sequential rotation across the entire reserved region. The library never reuses a block until all other blocks in the region have been used. This distributes erase cycles evenly, multiplying the effective write endurance by the total number of logical blocks in the region.

Power-loss safety is achieved through the atomic commit protocol. Every write operation follows a strict sequence in which the new block is marked as a "pre-commit" candidate during the write, and only becomes the active block when a final commit flag is atomically programmed. If power is lost at any point before the commit, the partial block is automatically discarded on the next power-up because its commit flag is not in the finalized state. The previously committed block remains intact and becomes the active block instead.

Byte-level granularity is provided through the descriptor-based parameter model. The application declares a set of named parameters with their sizes. EEMUL manages a complete payload snapshot containing all parameters. A write to any parameter is first applied to the RAM shadow buffer and compared against the current payload: if no byte has actually changed, the whole write is suppressed and no Flash activity occurs at all - this differential (no-op) suppression is the primary wear-reduction mechanism. When a change is present, the full payload is written into a fresh, already-erased block; during programming EEMUL skips only those alignment units that are entirely 0xFF, because an erased Flash unit already holds that value.

3. EEMUL Architecture

3.1. Repository Structure

The EEMUL middleware ships as five source files at the root of the repository. Each file has a clearly defined responsibility within the overall system.

Table 3-1. Repository File Structure

File	Purpose and Contents
eemul.h	The complete public API. Defines all types, enumerations, configuration macros with defaults, the eemul_handle_t runtime structure, and every public function declaration. This is the only header an application needs to include. It also contains extensive Doxygen documentation covering the entire design.
eemul.c	The full middleware implementation. Contains the internal logic for block scanning, atomic commit, wear leveling, bad-block handling, differential write detection, buffer management, and all public API functions.
eemul_port.h	The hardware abstraction layer (HAL) interface definition. Declares the extern eemul_port_ops global, which the user must define. Also serves as the include point for MCU-specific port files.
eemul_port_template.c	A self-documenting template implementation of the HAL port. Contains complete stub functions for erase_page, program, read, and (optionally) crc32, each with comments explaining exactly what the function must do and how to replace the stub with your MCU vendor SDK calls.
eemul_config_template.h	A template configuration header listing every tunable macro with its default value and a description of its effect. Copy this file to your project and rename it eemul_config.h, then adjust the values for your hardware.

3.2. The Reserved Flash Region

The foundation of EEMUL is a contiguous region of on-chip Flash that is reserved exclusively for emulation purposes and must not be used for code or other data. This region is characterized by two compile-time parameters: EEMUL_REGION_END_ADDR (the exclusive end address) and EEMUL_NUMBER_OF_FLASH_PAGES (the number of physical Flash pages). The start address is derived automatically as EEMUL_REGION_END_ADDR minus the total region size.

The region is conceptually divided into a sequence of logical blocks. A logical block is the smallest unit that EEMUL reads or writes as a whole. Each logical block consists of a fixed-size header followed immediately by a fixed-size payload. The header contains metadata used to validate and sequence the block; the payload contains the complete current value of every emulated parameter, laid out according to the descriptor.

Reserving the region in the linker script is a critical step. If the linker is not explicitly told to leave this region empty, the toolchain may place code or data there, which would overwrite stored parameters. The exact linker syntax differs between GCC (LD script) and IAR (ICF script) and is shown in detail in Section 6.2.

Note: A minimum of two Flash pages must be reserved (EEMUL_NUMBER_OF_FLASH_PAGES >= 2). This is enforced by a compile-time #error in eemul.h. The reason is that EEMUL must never erase the page containing the currently active block. With only one page, there would be nowhere to write new data while preserving the active block.

3.3. Block Layout Modes

EEMUL supports two physical layout modes that determine how logical blocks are mapped onto Flash pages. The choice significantly affects storage density, write endurance, and the maximum payload size.

In Page Mode (EEMUL_BLOCK_MODE_PAGE), each logical block occupies one or more complete physical Flash pages. The block size is rounded up to the nearest page boundary. This means that a 1 KB page with a 200-byte payload (header + payload = ~216 bytes after alignment) still uses the entire 1 KB page. Page Mode is the simplest layout and is most appropriate when the payload is large relative to the page size, or when simplicity of analysis is preferred over density.

In Sub-Block Mode (EEMUL_BLOCK_MODE_SUBBLOCKS), each physical Flash page is divided into multiple fixed-size slots, and each slot is one logical block. For example, if the total block size (header plus aligned payload) is 32 bytes and the Flash page is 1 KB, then 32 sub-blocks fit in a single page. This is far more efficient: the same 2 KB emulation region that holds only 2 blocks in Page Mode can hold 64 sub-blocks in Sub-Block Mode, multiplying the write endurance by a factor of 32.

The trade-off is that Sub-Block Mode requires the total block size (header + payload) to divide evenly into the page size. If the payload is large - for example, several hundred bytes - there may be very few sub-blocks per page, and the efficiency gain over Page Mode becomes marginal. For small parameter sets (total payload under 64 bytes), Sub-Block Mode is almost always the correct choice.

Table 3-2. Block Mode Comparison

Aspect	Page Mode	Sub-Block Mode
Block size	Rounded up to page boundary	header + payload, aligned to EEMUL_ALIGN_BYTES
Blocks per page	1 (always)	floor(page_size / block_bytes)
Best for	Large payloads (> half the page size)	Small payloads; maximizing endurance
Endurance multiplier	Equal to number of	Blocks per page x pages reserved

Aspect	Page Mode	Sub-Block Mode
	pages reserved	
Erase boundary	Block == page, simple	Multiple blocks share a page; only erase when entire page is consumed

3.4. The Block Header

Every logical block begins with a header that serves three purposes: identifying valid EEMUL blocks within the Flash region, ordering blocks by their write sequence, and flagging blocks as committed, uncommitted (pre-commit), or permanently bad. EEMUL provides two header formats selectable at compile time.

The Compact Header is 16 bytes and contains the minimum set of fields needed for correct operation. The magic field is the 32-bit ASCII constant "EMUL" (0x4C554D45). Any block whose first four bytes do not match this constant is immediately ignored - this protects against confusion with erased Flash (which reads as 0xFFFFFFFF) or random data from unrelated code sections. The sequence field is a 32-bit monotonically increasing counter. Among all valid blocks found during a region scan, the block with the highest sequence number is the most recently committed block and becomes the active block. The bad_marker field, normally 0xFFFFFFFF, is programmed to 0xBAD0BAD0 to permanently retire a block under the mark-on-flash policy (supported in both compact and full-header modes). The commit_flag is the most critical field: it is initialized to 0xFFFFFFFF at the start of a write, and only after the payload has been fully programmed is it atomically cleared to 0x00000000. A block is only considered committed and valid if magic matches, bad_marker is not the retirement value, and commit_flag is 0x00000000. The compact header does not carry an integrity CRC - if payload corruption detection is required, use the full header.

The Full Header is 32 bytes and extends the compact header with additional integrity fields. The magic constant is extended to the 64-bit ASCII string "EEMULATE" (0x4554414C554D4545). A payload_len field records the number of payload bytes so that a mismatch can be detected. A descriptor_crc field covers the parameter descriptor array, allowing EEMUL to detect if the firmware has been updated with a different parameter layout since the stored block was written. A bad_marker field, normally 0xFFFFFFFF, is programmed to 0xBAD0BAD0 when a block is permanently retired due to repeated hardware errors. The full header also includes a unified block_crc field: a CRC32 computed over the header fields preceding it (magic, sequence, payload_len, descriptor_crc) AND the entire payload. Because block_crc covers both header and payload bytes in a single check, in-place bit flips anywhere in the block are detected on the next read. The block_crc is left at the erased value (0xFFFFFFFF) at the start of a write and only programmed after the payload has been fully written, in the dedicated CRC step of the commit sequence (see Section 3.5).

Table 3-3. Block Header Field Reference

Field	Compact (16 B)	Full (32 B)	Description
magic	32-bit "EMUL"	64-bit	First field in every header. Checked

Field	Compact (16 B)	Full (32 B)	Description
		"EEMULATE"	before any other field. Blocks with wrong magic are silently skipped.
sequence	uint32_t	uint32_t	Monotonically increasing write counter. The block with the highest valid sequence is the active block.
payload_len	Not present	uint32_t	Expected payload size in bytes. Mismatch with computed payload_size triggers block rejection.
descriptor_crc	Not present	uint32_t	CRC of the parameter descriptor array at init time. Detects descriptor changes between firmware updates.
block_crc	Not present	uint32_t	CRC32 covering the 20 leading header bytes (magic, sequence, payload_len, descriptor_crc) AND the entire payload. Programmed together with descriptor_crc as one double-word after the payload write. Detects in-place corruption of either header or payload bytes.
bad_marker	0xBAD0BAD0 = retired	0xBAD0BAD0 = retired	Present in both header formats (offset 8 compact, offset 24 full). Written by the bad-block handler to permanently exclude a block across all power cycles.
commit_flag	0xFFFFFFFF = pre-commit; 0x00000000 = committed	Same	The atomic commit indicator. Only blocks with commit_flag == 0x00000000 AND all other checks passing are considered valid.

3.5. The Atomic Commit Protocol - Power-Loss Safety in Detail

The atomic commit protocol is the mechanism that guarantees EEMUL never presents corrupted or partial data to the application, regardless of when power is removed. The protocol is designed around the physical properties of Flash memory: bits can only transition from 1 to 0 (programming), and programming a word that is already 0x00000000 has no effect. The commit flag exploits this by using the final programming of a single word to atomically finalize a block.

A complete write operation proceeds through the following eight steps. Understanding each step also reveals exactly what happens if power is cut at that point:

Step 1 - Load shadow buffer: The active payload is read from Flash into the RAM shadow buffer (shadow_buf). This is a simple memory copy from the Flash address of the active block payload. The shadow buffer now holds the current state of all parameters.

Step 2 - Apply deltas in RAM: The requested parameter changes are applied to the shadow buffer. EEMUL then performs a byte-by-byte comparison of the modified shadow buffer against the old payload. If no bytes have actually changed, the write is aborted immediately and EEMUL_STATUS_OK is returned without touching Flash - this differential write detection is a key wear-reduction mechanism.

Step 3 - Select the next candidate block: EEMUL selects the next block in the rotation sequence. In Page Mode it skips any block that shares the same Flash page as the current active block (to avoid erasing the active data); in Sub-Block Mode multiple logical blocks intentionally share a page, so only the active block itself is avoided. Blocks marked bad are also skipped. If no safe candidate is available, the commit ultimately fails and eemul_write_param() / eemul_write_at() return EEMUL_STATUS_ERR_HW.

Step 4 - Erase the candidate page: If the candidate block is in Page Mode, or if it is the first sub-block in a page in Sub-Block Mode (meaning the page has been fully consumed and needs recycling), the page is erased. This is the most time-consuming step (typically 1-10 ms). If power is lost here, the active block is unaffected; on recovery, EEMUL scans and re-finds it.

Step 5 - Write pre-commit header: Only the pre-commit prefix of the header is programmed - 8 bytes for the compact header, or 16 bytes for the full header - covering magic, the new sequence number, and (in full-header mode) the payload length. The descriptor_crc and block_crc fields are deliberately left at the erased value (0xFFFFFFFF): they are programmed together later, in Step 7, after the payload exists, so that this 8-byte unit is written exactly once (a hard requirement of ECC Flash). The commit_flag also remains at its erased value (0xFFFFFFFF). This marks the block as a "pre-commit candidate." If power is lost here, the block is not yet valid. On recovery, EEMUL sees this block has magic matching and a sequence number, but commit_flag != 0x00000000, so it discards this block and falls back to the previous active block.

Step 6 - Program the payload: The full payload is written word-by-word from the shadow buffer into the Flash payload area of the candidate block. Each word is programmed one EEMUL_ALIGN_BYTES unit at a time. Align-units whose bytes are already all 0xFF are skipped - the slot is already in that state from the page erase, so a no-op program is unnecessary. This makes payload values that contain all-0xFF chunks (for example a uint64_t field equal to UINT64_MAX) round-trip safely on ECC-protected double-word Flash, where the FMC would otherwise reject an all-0xFF write. If power is lost here, the payload is partially written, but again, commit_flag is still 0xFFFFFFFF, so the block is not valid. The previous active block remains the active block.

Step 7 - Compute and write descriptor_crc and block_crc (full header only): EEMUL computes a CRC32 over the 20 leading header bytes (magic, sequence, payload_len, descriptor_crc) concatenated with the payload bytes just programmed. It then programs descriptor_crc and block_crc together as a single 8-byte double-word (DW2) into the still-erased unit - one program operation, never a re-program. If power is lost during this step, the double-word may be left partially programmed (some bits not yet flipped from 1 to 0), but commit_flag is still

0xFFFFFFFF, so the block remains a pre-commit candidate and is discarded on recovery. In compact-header mode (EEMUL_USE_FULL_BLOCK_HEADER = 0) this step is skipped entirely - the compact header carries no integrity CRC.

Step 8 - Atomic commit: commit_flag is programmed to 0x00000000. This is a single Flash programming operation on one word. Either it completes and the block becomes the new active block, or power is lost first and it does not complete. There is no intermediate state: a word program either writes the full word or does not. After this step, the candidate block passes all validity checks and is the new active block.

This eight-step sequence guarantees that at any point of power loss, the system can always recover to a fully valid state. The critical insight is that the commit flag is the last thing written, and it is a single programming operation - which is atomic at the hardware level. In full-header mode, descriptor_crc and block_crc are written together immediately before commit_flag, so a successfully committed block always carries a CRC that has been verified against the actually-programmed header and payload bytes.

3.6. Wear Leveling - Linear Rotation

EEMUL uses linear rotation as its wear leveling strategy. The library tracks the index of the most recently written block (active_block_index). When a new write is needed, the next candidate is active_block_index + 1, wrapping around to 0 after the last block. This ensures that all blocks are used in strict round-robin order before any block is reused.

This approach has two important properties. First, it guarantees that no block receives a second erase cycle until every other block in the region has received at least one - the maximum spread of wear. Second, it is extremely simple to implement and requires no RAM-based wear counters per block, because the sequence numbers embedded in each block header serve as an implicit timestamp of last use.

The practical effect of wear leveling is that the write endurance of the emulation region scales linearly with the number of logical blocks. A region with 64 sub-blocks supports 64 times as many application writes as a single Flash page, so the 10,000-cycle Flash endurance rating becomes 640,000 application-level write operations before any page reaches its limit.

3.7. Bad-Block Handling

Hardware defects can cause individual Flash pages to fail erase or program operations. Without bad-block handling, a single failing page can permanently prevent new writes from being committed. EEMUL provides two configurable policies for managing bad blocks.

The Retry-Only policy (EEMUL_BADBLOCK_POLICY_RETRY) maintains a per-block error counter in RAM. When a block experiences a hardware failure, its error counter is incremented. If the counter reaches bad_retry_threshold, the block is flagged in the in-RAM bad_block_flags array and skipped for the remainder of the current power session. On the

next power-up, the in-RAM flags are reset and EEMUL will attempt to use the block again. This policy is appropriate when transient errors are possible (e.g., from ESD events) and the block may recover.

The Mark-on-Flash policy (EEMUL_BADBLOCK_POLICY_MARK) extends the retry-only behavior: when a block reaches bad_retry_threshold errors, EEMUL also writes the bad_marker value (0xBAD0BAD0) into the block header on Flash. This marker is detected during every subsequent region scan, permanently excluding the block from use across all power cycles. The bad_marker field is present in both header formats (offset 8 in the compact header, offset 24 in the full header), so this policy works in compact and full-header modes alike. Use this policy when hardware defects are expected to be permanent.

3.8. The Shadow RAM Buffer and Differential Writes

Every EEMUL handle maintains a shadow buffer in RAM (shadow_buf) whose size equals the total aligned payload size. The shadow buffer always contains the most recently committed values of all parameters. When EEMUL is initialized, the shadow buffer is populated by reading the active block payload from Flash. The shadow buffer is the staging area for differential writes and batch updates (described below); it is not the read path. Reads are served directly from the active block in memory-mapped Flash, which on GD32 devices is as fast as reading SRAM and adds no Flash bus penalty.

When a write is requested, EEMUL refreshes the shadow buffer from the current active block in Flash and compares the requested new values against it byte by byte. If every byte in the affected range is unchanged, the write is suppressed entirely: EEMUL returns EEMUL_STATUS_OK without consuming a write cycle. This differential write detection is particularly valuable in applications that periodically "save" all parameters regardless of whether they have changed - for example, a periodic configuration save triggered by a timer.

After a successful commit, the shadow buffer reflects the newly stored values. Because each write first refreshes the shadow buffer from Flash before diffing, the buffer is re-synchronized with the on-Flash state on every write.

3.9. Sequence Numbers and Recovery

Every committed block carries a 32-bit sequence number that increases monotonically with each write. The sequence number serves as the ordering key during region scan: when multiple valid (committed) blocks are found in the region, the one with the highest sequence number is selected as the active block. Pre-commit blocks (commit_flag != 0x00000000), blocks with invalid magic, and - in full-header mode - blocks whose block_crc does not match the recomputed header+payload CRC are silently ignored.

The enable_monotonic_sequence configuration option controls how sequence numbers behave across power cycles and region reformats. When set to true (recommended), the library initializes next_sequence to active_sequence + 1 after recovery, ensuring strict

monotonic ordering even after the region has been formatted and reinitialized. When set to false, the sequence number resets to 1 after a format, which can theoretically cause ambiguity if the highest and lowest sequence numbers are both present in the region after wrap-around. Monotonic mode is safer and should be used unless ROM constraints make it necessary to avoid the extra RAM used to track next_sequence.

The recovery procedure (eemul_recover) performs a full linear scan of all blocks in the region. It reads each block header, validates magic, payload_len, descriptor_crc, bad_marker and commit_flag. In full-header mode it then also reads the payload and recomputes the combined header+payload CRC, comparing it against block_crc - any mismatch discards the block. Among the surviving valid blocks the one with the highest sequence number wins. Recovery also collects bad_marker values from the full header to restore permanent bad-block flags. The recovered active block is then loaded into the shadow buffer. If no valid block is found - for example, on first boot with a blank Flash region - EEMUL performs an auto-format: it erases all pages in the region and writes a first committed block whose payload is zero-initialized (all 0x00), giving a deterministic cold-start state for all parameters.

4. Configuration Reference

All compile-time configuration is controlled through macros defined in eemul_config.h. The library ships with eemul_config_template.h containing every macro with its default value. Copy this file into your project include path, rename it eemul_config.h, and adjust the values for your specific hardware and application requirements. The macros are processed by eemul.h before any other definitions, so they take effect throughout the entire library.

Table 4-1. Configuration Macro Reference

Macro	Default	Description and Design Guidance
EEMUL_REGION_END_ADDR	0x08010000	Exclusive end address of the emulation region. The region ends at (but does not include) this address. Must be aligned to a Flash page boundary. The start address is derived as end - (pages x page_size). Example: placing the region in the last 2 KB of a 64 KB Flash gives end = 0x08010000, start = 0x0800F800.
EEMUL_NUMBER_OF_FLASH_PAGES	2	Number of physical Flash pages in the reserved region. The minimum enforced by the library is 2 (a compile-time error is generated if fewer than 2 are specified). More pages = more logical blocks = higher endurance. Balance endurance requirements against available Flash budget.
EEMUL_FLASH_PAGE_SIZE	0x400 (1 KB)	Physical page size in bytes. Must match the erase granularity of the target part. GD32 page sizes range from a few hundred bytes on small-capacity parts to tens of kilobytes on large-capacity parts; consult the FMC chapter of the device user manual for the exact value.
EEMUL_ALIGN_BYTES	4	Flash programming alignment and granularity in bytes. All write operations are performed in units of this size. Supported values are 2 (half-word), 4 (word), and 8 (double-word). The chosen value must match the FMC programming unit of the target part: ECC-protected double-word Flash mandates 8, while half-word and word FMCs accept the smaller values within the limits of the hardware.

Macro	Default	Description and Design Guidance
		Mismatches cause undefined behavior; the port file should reject unsupported combinations at compile time via #error.
EEMUL_ENABLE_DYNAMIC_ALLOC	0	Buffer allocation strategy. 0 = all buffers are static arrays allocated at compile time (no heap required, suitable for safety-critical or deeply constrained devices). 1 = buffers are allocated with malloc/calloc at eemul_init() time, sized to the actual descriptor. Set to 1 when multiple EEMUL instances with different descriptors are used, or when RAM is tightly constrained and the static maximum sizes waste too much RAM.
EEMUL_MAX_PAYLOAD_BYTES	128	Maximum supported payload size in bytes, used for static buffer sizing. Must be >= the sum of all parameter sizes in your descriptor. With static allocation (EEMUL_ENABLE_DYNAMIC_ALLOC = 0) it sizes the fixed payload and CRC buffers, and eemul_init() returns EEMUL_STATUS_ERR_LAYOUT if the descriptor payload exceeds it. With dynamic allocation this macro is ignored: every buffer, including the full-header CRC scratch buffer, is sized to the actual descriptor at init time.
EEMUL_ENABLE_PRECOMPUTE_PARAM_OFFSETS	1	Parameter offset caching. 1 = at init time, EEMUL computes the byte offset of each parameter within the payload and stores it in a lookup table. Subsequent read and write operations use O(1) offset lookup. 0 = offsets are recomputed on each access by iterating the descriptor, saving RAM at the cost of a few extra cycles per access. For most applications, keep this enabled.
EEMUL_MAX_PARAM_COUNT	32	Maximum number of parameters in the descriptor, used only for sizing the static offset table when EEMUL_ENABLE_PRECOMPUTE_PARAM_OFFSETS = 1. Increase if your descriptor contains more than 32

Macro	Default	Description and Design Guidance
		parameters.
EEMUL_USE_FULL_BLOCK_HEADER	0	0 = compact 16-byte header (magic "EMUL", sequence, bad_marker, commit_flag). Carries no integrity CRC; minimizes header overhead and maximizes sub-block density per page. 1 = full 32-byte header (magic "EEMULATE", sequence, payload_len, descriptor_crc, block_crc, bad_marker, commit_flag). The block_crc field covers the preceding header fields AND the entire payload, giving end-to-end integrity protection. Required for header+payload corruption detection, permanent bad-block marking, and descriptor change detection. Recommended for production systems.

The following derived values are computed automatically from the above macros and are available for use in application code:

Table 4-2. Derived Configuration Values

Derived Macro	Formula and Meaning
EEMUL_REGION_SIZE_BYTES	EEMUL_NUMBER_OF_FLASH_PAGES x EEMUL_FLASH_PAGE_SIZE. Total size of the reserved Flash region in bytes.
EEMUL_REGION_START_ADDR	EEMUL_REGION_END_ADDR - EEMUL_REGION_SIZE_BYTES. Inclusive start address of the reserved region.
EEMUL_BLOCK_HEADER_MAGIC	0x4C554D45 ("EMUL") in compact mode, 0x4554414C554D4545 ("EEMULATE") in full header mode. Written into every block header. Guarded with #ifndef in eemul.h, so it is configurable with a default value - an application may override it in eemul_config.h.
EEMUL_BLOCK_HEADER_BAD_MARK_VALUE	0xBAD0BAD0. The value written into the bad_marker field of a block header (compact or full) to permanently retire a block under the mark-on-flash policy.

5. Porting to GD32 MCUs

5.1. Understanding the HAL Port Architecture

EEMUL is designed to be completely independent of any specific MCU or Flash driver library. It interacts with Flash hardware exclusively through a set of function pointers collected in the eemul_port_ops_t structure. This structure is defined in eemul.h and is passed to eemul_init() during initialization. The library stores a pointer to this structure in the handle and calls the functions directly whenever it needs to erase, program, or read Flash.

The separation between the middleware logic and the hardware driver has a practical consequence: porting EEMUL to a new GD32 target requires only implementing four functions (three mandatory, one optional) in a new C source file. No library source files need to be modified. The template file eemul_port_template.c provides a skeleton with comments explaining exactly what each function must do.

Table 5-1. HAL Port Function Interface

Function	Required	Signature	Contract and Implementation Notes
erase_page	Yes	bool (*)(uint32_t addr)	Erase one physical Flash page at addr. The addr parameter is always page-aligned. Must return true if the erase succeeded, false if the hardware reported an error. Must block until the erase is complete - do not return before the hardware confirms completion. The implementation calls the appropriate FMC erase function for the target part (fmc_page_erase for page-based FMCs, fmc_sector_erase for sector-based FMCs) and checks the returned fmc_state_enum.
program	Yes	bool (*)(uint32_t addr, const void *src)	Program exactly EEMUL_ALIGN_BYTES bytes from src to the Flash address addr. The addr is always EEMUL_ALIGN_BYTES-aligned. Must return true on success, false on hardware error. For word programming (EEMUL_ALIGN_BYTES=4): copy

Function	Required	Signature	Contract and Implementation Notes
			src to a uint32_t and call fmc_word_program(). For double-word programming (EEMUL_ALIGN_BYTES=8): call fmc_doubleword_program().
read	Yes	void (*)(uint32_t addr, void *dst, uint32_t len)	Read len bytes from the Flash address addr into dst. On GD32 (and most ARM Cortex-M MCUs), internal Flash is memory-mapped and directly addressable by the CPU. This function is therefore always implemented as: memcpy(dst, (const void*)addr, len). No return value because a read failure is not possible on memory-mapped Flash.
crc32	Optional	uint32_t (*)(const void *data, uint32_t len)	Compute a CRC32 checksum over len bytes of data. Only required when EEMUL_USE_FULL_BLOCK_HEADER = 1. If set to NULL in the eemul_port_ops_t structure, EEMUL automatically uses a built-in software CRC32 implementation. If the target device includes a hardware CRC unit, connecting it here improves throughput.

5.2. FMC Unlock and Lock - A Critical Requirement

GD32 Flash Memory Controller (FMC) requires a specific unlock sequence before any erase or program operation can be performed. The unlock sequence writes two specific keys (UNLOCK_KEY0 and UNLOCK_KEY1) to the FMC_KEY register. After the operation is complete, the FMC should be locked again by setting the LK bit in FMC_CTL. This is essential for two reasons: it protects the Flash from accidental modification by other code, and it prevents runaway pointer bugs from corrupting stored parameters.

Both the erase_page and program HAL functions must follow this pattern: unlock, perform the operation, lock. The standard GD32 peripheral library provides fmc_unlock() and fmc_lock() functions for this purpose. The erase and program functions in the port layer should also clear any pending FMC status flags before starting the operation using fmc_flag_clear(), as leftover flags from a previous operation may cause the hardware to reject

the new operation.

Note: Never attempt to execute code from a Flash page that is being erased or programmed. Execution must come from a different Flash bank, from SRAM, or from the instruction cache during these operations. On single-banked devices the CPU must therefore be executing code that runs entirely from cache or SRAM during FMC operations, or the FMC wait states must be correctly configured. Refer to your device user manual for details.

5.3. GD32F50x Complete Port Implementation

The GD32F50x series uses 2 KB (bank0) or 4 KB (bank1) Flash pages with half-word or word programming granularity. The following complete port implementation uses the GD32F50x standard peripheral library. It handles the FMC unlock/lock sequence, clears the bank-specific status flags, and returns the correct boolean result based on the FMC state after each operation.

Table 5-2. GD32F50x HAL Port Implementation

```
/*
 * 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. Considerations for Sector-Based Flash

Some GD32 parts use a sector-based Flash architecture in which the erase granularity varies across the address space - typically a few small sectors at the start of Flash followed by progressively larger ones. EEMUL_FLASH_PAGE_SIZE must equal the physical erase unit of the sectors that hold the emulation region, so the reserved region must lie entirely within sectors of a single size.

It is strongly recommended to reserve the emulation region within a single sector size. If two 16 KB sectors are reserved, set EEMUL_FLASH_PAGE_SIZE = 0x4000 (16 KB). With a small payload (say, 60 bytes total), a 32-byte block in Sub-Block Mode gives floor(16384 / 32) = 512 sub-blocks per page, and with 2 pages, that is 1024 total blocks - over 10 million writes before any page reaches its endurance limit.

The FMC API for sector-based parts typically uses fmc_sector_erase(sector_number) instead of fmc_page_erase(address). The port implementation must therefore translate the Flash address passed by EEMUL into a sector number, either with a small lookup helper or by computing the sector ID from the known sector layout of the target part.

Note: EEMUL_ALIGN_BYTES on sector-based parts is determined by the programming granularity of the FMC, not by the sector layout - typically 4 (word) or 8 (double-word). Check the FMC chapter of the device user manual to confirm the supported value.

5.5. Considerations for Double-Word Programming and ECC Flash

Some GD32 parts use 64-bit double-word programming as the native programming granularity, often combined with hardware ECC (Error Correction Code) over each double-word in Code Flash. These features require specific attention when configuring EEMUL.

Set EEMUL_ALIGN_BYTES = 8 on such targets. All EEMUL write operations will then operate in 8-byte units, which matches the double-word programming constraint of the FMC hardware. The payload size and header size are aligned to 8-byte boundaries. This slightly increases per-block overhead for small payloads but is required for hardware correctness.

ECC operates at the hardware level and is largely transparent to EEMUL. Single-bit errors

are corrected automatically by the hardware on read; double-bit errors are reported as a bus fault. These are handled at the system level and do not require special treatment in the EEMUL port layer. Two ECC behaviors do interact with EEMUL directly: a Flash write that does not cover the full double-word width raises an ECC error (which EEMUL satisfies automatically because every program operation is exactly EEMUL_ALIGN_BYTES = 8 bytes), and a program of an already-erased double-word is rejected by the FMC because no ECC syndrome can be generated from an all-0xFF write. EEMUL handles the latter by skipping the port program() call when the source align-unit is already all 0xFF, so a payload that legitimately contains 0xFF chunks (an uninitialized field, a uint64_t equal to UINT64_MAX) round-trips correctly without triggering an ECC fault.

6. Step-by-Step Integration Guide

This section walks through a complete integration of EEMUL into a GD32F50x project from start to finish. Each step is explained with the reasoning behind it, not just the mechanics. The example target is a GD32F503VG with 1024 KB Flash and 96 KB SRAM, using Keil MDK with the GD32F50x Standard Peripheral Library.

6.1. Step 1 - Add Source Files to the Project

Copy the following files from the EEMUL repository into your project directory. A recommended layout is to place them in a Middleware/EEMUL/ subdirectory:

eemul.h - the public header; must be on the compiler include path

eemul.c - the middleware implementation; must be compiled and linked

eemul_port.h - the HAL interface header

eemul_port_template.c - copy this, rename it (e.g., eemul_port_gd32f50x.c), and fill in your hardware implementation

eemul_config_template.h - copy this, rename it to eemul_config.h, and adjust the values

In Keil MDK: right-click the target group in the Project window and select "Add Existing Files". Add eemul.c and your port .c file. Add the Middleware/EEMUL/ directory to the compiler include paths under Project -> Options for Target -> C/C++ -> Include Paths.

6.2. Step 2 - Reserve the Flash Region in the Linker Script

The linker must be told to leave the emulation region empty. If this step is skipped, the linker may place code, read-only data, or the interrupt vector table inside the reserved region, which EEMUL will overwrite during normal operation.

For GD32F503VG with 1024 KB Flash, placing the emulation region in the last 8 KB (using two 4 KB pages of bank1) leaves 1016 KB for code and read-only data - more than adequate for most applications. The configuration values for this layout are: EEMUL_REGION_END_ADDR = 0x08100000 (end of 1024 KB Flash from 0x08000000), EEMUL_NUMBER_OF_FLASH_PAGES = 2, EEMUL_FLASH_PAGE_SIZE = 0x1000.

Table 6-1. Linker Script Modification for GD32F503VG (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
}
```

For Keil MDK scatter file (ARM Linker), the equivalent declaration places the region in the scatter description. In IAR EWARM (.icf), the corresponding construct is a region declaration with the do not initialize attribute. In all cases, verify that the reserved region does not overlap with any other section in the linker map file (usually named project_name.map) after building.

Note: After modifying the linker script, perform a full clean and rebuild of the project. Inspect the .map file to confirm that no code or data sections are placed between EEMUL_REGION_START_ADDR and EEMUL_REGION_END_ADDR. Also verify that the Flash programming algorithm used by your debugger correctly skips this region and does not erase it during download - most CMSIS-Pack algorithms do this automatically when the linker leaves the region unoccupied.

6.3. Step 3 - Create eemul_config.h

Copy eemul_config_template.h to your project include directory and rename it to eemul_config.h. Edit the values to match your hardware and requirements. The following example is configured for a GD32F503VG with a small parameter set (total payload under 64 bytes), using Sub-Block Mode for maximum endurance and compact headers to maximize block density.

Table 6-2. eemul_config.h for GD32F503VG (Sub-Block Mode, Compact Header)

```
/*
 * 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 */
```

With this configuration, the library derives its layout from the actual descriptor payload - not from EEMUL_MAX_PAYLOAD_BYTES, which only bounds the static buffer size. The descriptor defined in Step 4 totals 44 bytes, so the derived layout is: total region = 2 x 4096 = 8192 bytes; block size = header (16 bytes) + payload aligned to a 4-byte boundary (16 + 44 = 60 bytes, already aligned); sub-blocks per page = floor(4096 / 60) = 68; total blocks = 2 pages x 68 sub-blocks = 136 blocks; write endurance = 136 x 10,000 = 1,360,000 application writes before any page reaches its rated endurance limit.

6.4. Step 4 - Define the Parameter Descriptor

The parameter descriptor is the core of the EEMUL application interface. It is an array of uint8_t values where each element specifies the size in bytes of one parameter. Parameters are identified by their zero-based index within this array, referred to as the parameter ID.

Good descriptor design groups logically related parameters together and sizes them to match the actual data types used in the application. It is important to use sizeof() rather than hard-coded constants, since the compiler will catch type size changes automatically. If a parameter is later expanded - for example, a calibration table grows from 16 bytes to 32 bytes - the descriptor must be updated and the eemul_config.h EEMUL_MAX_PAYLOAD_BYTES value must be increased to match.

The following example defines a realistic parameter set for a motor control application: firmware version (for OTA tracking), calibration coefficients, an error log, an operational cycle counter, and active control parameters.

Table 6-3. Parameter Descriptor Definition Example

```
/*
 * 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. Step 5 - Initialize the Middleware

EEMUL initialization is the most important step. The call to eemul_init() performs several sequential operations: it validates the input parameters and configuration, allocates or binds runtime buffers, verifies the Flash region geometry, scans the emulation region for the latest valid committed block, and if none is found (first boot), performs an auto-format. Each of

these sub-operations can fail independently, and the returned status code identifies which step failed.

The initialization configuration struct (eemul_init_config_t) must be fully populated before calling eemul_init(). Each field has behavioral implications that should be understood before setting a value. The badblock_policy field determines what happens when a Flash block fails to erase or program. EEMUL_BADBLOCK_POLICY_MARK is recommended for production designs because it permanently avoids bad blocks across reboots. EEMUL_BADBLOCK_POLICY_RETRY is appropriate during development when you want to retry blocks that may have failed transiently during debugging sessions. The bad_retry_threshold field sets how many hardware errors a block must accumulate before being retired - a value of 2 or 3 is a reasonable balance between tolerance for transient errors and prompt retirement of genuinely defective blocks. Setting enable_monotonic_sequence to true is always recommended: it prevents sequence number ambiguity after region wraps.

Table 6-4. EEMUL Initialization with Full Error Handling

```
/*
 * 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. Step 6 - Reading Parameters

Reading a parameter with eemul_read_param() copies the parameter value from the active block in memory-mapped Flash into the destination buffer provided by the application. On GD32 devices Flash is mapped into the CPU address space, so a read is a simple memory copy that executes in a few cycles with no bus stalls, no blocking, and no erase/program interaction. The active block always holds the most recently committed values.

The size of the destination buffer must exactly match the declared size of the parameter in the descriptor. EEMUL validates this match at runtime and returns EEMUL_STATUS_ERR_PARAM if there is a mismatch. Using the same typedef for both the descriptor entry and the destination buffer (as shown in the descriptor example) ensures the sizes stay in sync automatically.

Table 6-5. Reading Parameters - Examples

```
/* -- 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. Step 7 - Writing Parameters (Single and Atomic)

Writing a single parameter with eemul_write_param() triggers the full eight-step atomic commit sequence described in Section 3.5. The call blocks until either the new block has been successfully committed to Flash, or an error has occurred. On return, the shadow buffer has been updated to reflect the new value if the write succeeded.

The differential write detection means that calling eemul_write_param() with data identical to the current stored value is safe and efficient - the library detects the match and returns EEMUL_STATUS_OK immediately without any Flash activity. Applications can therefore call eemul_write_param() unconditionally during periodic save routines without fear of wearing out the Flash prematurely.

Error handling for writes is important. EEMUL_STATUS_ERR_HW indicates a Flash hardware failure during erase or program. In this case, EEMUL has already attempted the write, and the block policy has updated the bad-block state. The application should typically log the error and continue - the previous active block is still valid. The same EEMUL_STATUS_ERR_HW code is also returned when no safe candidate block is available (all remaining blocks are bad or unsafe), which indicates that the emulation region is exhausted or severely degraded. The write API does not return EEMUL_STATUS_ERR_NOBLOCK; that code originates only from the eemul_init() / eemul_recover() format path.

Table 6-6. Writing Parameters - Examples and Error Handling

```
/* -- 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. Step 8 - Batch Updates (Multiple Parameters, One Commit)

A batch update allows multiple parameters to be changed atomically within a single Flash write cycle. This is essential when two or more parameters are logically coupled - for example, a calibration coefficient and its associated checksum, or a control parameter set where all values must be consistent with each other. If a single eemul_write_param() call were used for each parameter individually, a power failure between calls would leave the system with a partially updated state where some parameters hold old values and others hold new ones.

A batch update works by staging all changes in the shadow buffer without committing them, then committing the entire shadow buffer as one new block. The sequence is: call eemul_begin_batch() to load the active block's committed payload into the shadow buffer, establishing a known baseline; call eemul_update_param_in_shadow() for each parameter to be changed; call eemul_commit_batch() to write the complete updated shadow buffer to Flash as a new block. If eemul_commit_batch() fails, none of the new values are committed - the previous block remains the active block.

A batch update also provides a performance advantage: instead of n separate Flash write cycles for n parameter changes, a single write cycle is used regardless of the number of parameters updated. On Flash hardware where each write cycle takes several milliseconds, this can significantly reduce the time the CPU spends blocked in Flash operations.

Table 6-7. Batch Update - Atomic Multi-Parameter Commit

```
/*
 * 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 */
}
```

After eemul_commit_batch() returns successfully, the newly written block becomes the active block. Subsequent reads via eemul_read_param() will return the new values immediately.

7. Design Decisions and Practical Recommendations

7.1. Choosing the Right Block Mode

The choice between Page Mode and Sub-Block Mode is the most consequential configuration decision. The decision rule is straightforward: if the total payload size (header + aligned payload) is larger than half the page size, use Page Mode. Otherwise, use Sub-Block Mode. The reason is that Sub-Block Mode provides a density benefit only when multiple sub-blocks fit within one page. If the payload is so large that only one sub-block fits per page, Sub-Block Mode has no advantage over Page Mode - and introduces slightly more bookkeeping overhead.

For the vast majority of EEPROM emulation use cases - storing device configuration, calibration data, counters, and error logs - the payload is well under 200 bytes, and the page size is 1 KB or larger. Sub-Block Mode is almost always the correct choice in these cases.

7.2. Sizing the Emulation Region for Your Endurance Requirement

To size the emulation region, work backwards from your endurance requirement. Start by estimating the write rate: how many times per hour (or per day) does the application call eemul_write_param() or eemul_commit_batch()? Multiply by the expected product lifetime in hours. This gives the required number of application-level write operations.

Next, compute the number of logical blocks in your proposed configuration: total_blocks = (number_of_pages x page_size) / block_size in Sub-Block Mode, or total_blocks = number_of_pages in Page Mode. Each Flash page supports approximately 10,000 erase cycles, so the total Flash-level endurance equals total_blocks x 10,000. If this number exceeds your required application-level writes, the configuration is sufficient.

Table 7-1. Endurance Calculation Examples

Scenario	Calculation and Result
Industrial sensor, writes once per minute, 10-year lifetime	Required writes: 60 x 24 x 365 x 10 = 5,256,000. Sub-Block Mode, 1 KB pages, 48-byte blocks: 21 sub-blocks/page x 4 pages = 84 blocks. Endurance = 84 x 10,000 = 840,000 writes. INSUFFICIENT. Increase to 8 pages: 168 blocks x 10,000 = 1,680,000. Still insufficient. Solution: use two 16 KB sectors (EEMUL_FLASH_PAGE_SIZE = 0x4000). With 48-byte blocks: floor(16384 / 48) = 341 sub-blocks/page x 2 pages = 682 blocks. Endurance = 682 x 10,000 = 6,820,000 writes. SUFFICIENT with 1.3x margin.
Consumer device, writes once per hour, 5-year lifetime	Required writes: 24 x 365 x 5 = 43,800. Sub-Block Mode, 1 KB pages, 80-byte blocks: 12 sub-blocks/page x 2 pages = 24 blocks.

Scenario	Calculation and Result
	Endurance = 24 x 10,000 = 240,000 writes. SUFFICIENT with large margin (5.5x).
Safety device, writes once per power cycle, 20-year lifetime, 100,000 power cycles	Required writes: 100,000. Sub-Block Mode, 1 KB pages, 48-byte blocks: 21 sub-blocks/page x 2 pages = 42 blocks. Endurance = 42 x 10,000 = 420,000 writes. SUFFICIENT with 4.2x margin.

7.3. Full Header vs Compact Header

Use the full header in production designs unless ROM or block density constraints make it impractical. The full header provides: payload length validation (catches blocks with garbled length fields) and descriptor CRC checking (detects firmware updates that changed the parameter layout without erasing the region). These checks add 3-4 ms to the init time (from the additional Flash reads) but cost nothing at runtime. (Note that the mark-on-flash bad-block policy works in either header format - see Section 3.7.)

The compact header is appropriate during early development, in prototypes where the parameter layout is still changing frequently, or in extremely space-constrained environments where maximizing sub-block density is critical. Always migrate to the full header before finalizing production firmware.

Note: The bad_marker field is only written to Flash when EEMUL_BADBLOCK_POLICY_MARK is used (in either header format). With EEMUL_BADBLOCK_POLICY_RETRY, the bad_marker field is present in the header structure but is never written by the bad-block handler. When EEMUL_USE_FULL_BLOCK_HEADER = 1, the full header remains useful in this case for its payload_len and descriptor_crc checking.

7.4. Initializing Default Parameter Values

When EEMUL formats a blank region for the first time (auto-format on first boot), the payload of the initial block is zero-initialized (all 0x00) for a deterministic cold-start state. This means that on first boot, eemul_read_param() will return buffers filled with 0x00 unless the application explicitly writes default values.

The correct pattern is to write default values immediately after eemul_init() on first boot. The simplest way to detect first boot is to check a specific parameter that would never legitimately be zero - for example, a firmware version number whose major byte is always 1 or greater. If the major version byte reads as 0x00, the region was just formatted and defaults should be written.

Table 7-2. First-Boot Default Value Initialization Pattern

```
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 Reference

8.1. Status Codes

All EEMUL API functions return an eemul_status_t value indicating success or the specific type of failure. Applications should always check the return value of every API call - particularly write operations, which can fail due to transient or permanent Flash hardware errors.

Table 8-1. eemul_status_t Return Codes

Code	Value	Meaning and Recommended Response
EEMUL_STATUS_OK	0	Operation completed successfully. For write operations, either the data was committed to Flash or the write was suppressed because the data was unchanged (differential detection).
EEMUL_STATUS_ERR_PARAM	1	Invalid argument or configuration error. Causes include: NULL handle or pointer, out-of-range parameter ID, size argument does not match descriptor, or malloc failure when dynamic alloc is enabled. This error indicates a firmware bug and should be treated as fatal during development.
EEMUL_STATUS_ERR_HW	2	Flash hardware operation failed. The erase_page or program HAL function returned false. The bad-block error counter for the failing block has been incremented. The previous active block remains valid. Log the error and continue - do not treat this as fatal unless the system requires guaranteed write success.
EEMUL_STATUS_ERR_LAYOUT	3	Invalid Flash layout. The block size derived from the configuration macros and descriptor exceeds the page size, the payload exceeds EEMUL_MAX_PAYLOAD_BYTES, or another geometric inconsistency was detected. Check EEMUL_FLASH_PAGE_SIZE, EEMUL_ALIGN_BYTES, EEMUL_MAX_PAYLOAD_BYTES, and the total descriptor payload size. This is always a configuration error - it cannot occur at runtime if the configuration was correct at init time.
EEMUL_STATUS_ERR_NOBLOCK	4	No usable block found. Returned by eemul_init() / eemul_recover() when no committed block exists during recovery and the auto-format also fails. Runtime write and batch-commit failures - including the case

Code	Value	Meaning and Recommended Response
		where no candidate block is available - are reported as EEMUL_STATUS_ERR_HW, not this code. This is a severe condition indicating a fully exhausted emulation region or widespread Flash failure. Increase the emulation region size or handle as a system-level fault.

8.2. Initialization and Lifecycle Functions

Table 8-2. Initialization and Lifecycle API

Function	Returns	Description
eemul_init(handle, port_ops, init_config, param_descriptor, param_count)	eemul_status_t	Initialize the handle. Validates all inputs, binds the HAL, precomputes geometry and offsets, scans Flash for the active block, and auto-formats if no valid block is found. Must be called before any other API function. The handle must remain valid (not stack-allocated in a function that returns) for the lifetime of the EEMUL session.
eemul_deinit(handle)	void	Deinitialize the handle. Releases any dynamically allocated buffers if EEMUL_ENABLE_DYNAMIC_ALLOC = 1. After this call, the handle must be re-initialized before use. In static allocation mode, this function is effectively a no-op but clears internal state.
eemul_recover(handle)	eemul_status_t	Re-scan Flash and recover the active block. Useful after an external Flash operation (such as a firmware OTA update) that may have changed the emulation region contents. Also called internally during init. Reloads bad-block markers from full headers and restores sequence tracking.

8.3. Read and Write Functions

Table 8-3. Read and Write API

Function	Returns	Description
eemul_read_param(handle, param_id, dst)	eemul_status_t	Copy the current value of parameter param_id from the active block in memory-mapped Flash into dst. dst must point to a buffer of exactly the size declared for param_id in the descriptor. On GD32 the read is a direct memory copy with no bus penalty. Returns EEMUL_STATUS_ERR_PARAM if param_id is out of range or dst is NULL.
eemul_write_param(handle, param_id, src)	eemul_status_t	Atomically commit a new value for parameter param_id. Applies the differential write detection: if the new value is identical to the current shadow buffer value, returns EEMUL_STATUS_OK immediately without writing Flash. Otherwise executes the full eight-step atomic commit. Returns EEMUL_STATUS_ERR_HW on Flash failure or when no candidate block is available.
eemul_read_at(handle, offset, dst, len)	eemul_status_t	Read len bytes from the active block payload (memory-mapped Flash) starting at byte offset within the full payload image. Offset is an absolute byte position within the payload, not a parameter ID. Use eemul_get_param_offset() to obtain the offset for a specific parameter. Useful for reading sub-fields without knowing the full parameter structure.
eemul_write_at(handle, offset, src, len)	eemul_status_t	Write len bytes to the payload at byte offset (atomic commit). Same differential write detection applies. Useful for updating a sub-field of a

Function	Returns	Description
		large parameter without reading and rewriting the entire parameter.

8.4. Batch Update Functions

Table 8-4. Batch Update API

Function	Returns	Description
eemul_begin_batch(handle)	eemul_status_t	Prepare for a multi-parameter batch update. Loads the active block's committed payload from Flash into the shadow buffer, establishing a known baseline. After this call, the shadow buffer can receive multiple parameter updates via eemul_update_param_in_shadow() before a single eemul_commit_batch(). Returns EEMUL_STATUS_ERR_PARAM if handle is NULL or invalid.
eemul_update_param_in_shadow(handle, param_id, src, len)	eemul_status_t	Stage a parameter update in the shadow buffer without committing to Flash. len must equal the declared size of param_id in the descriptor. Can be called multiple times between eemul_begin_batch() and eemul_commit_batch(). Returns EEMUL_STATUS_ERR_PARAM on size mismatch or invalid ID.
eemul_commit_batch(handle)	eemul_status_t	Commit all staged shadow buffer changes to Flash as a single atomic block. If power fails during this call, none of the staged changes are committed - the previous active block remains valid. On success, the shadow buffer reflects all staged changes. Returns EEMUL_STATUS_ERR_HW on

Function	Returns	Description
		Flash failure or when no candidate is available.

8.5. Utility Functions

Table 8-5. Utility and Introspection API

Function	Returns	Description
eemul_get_payload_size(handle)	uint16_t	Returns the total aligned payload size in bytes, computed from the descriptor at init time. Useful for logging, diagnostics, and verifying that EEMUL_MAX_PAYLOAD_BYTES is correctly sized.
eemul_get_param_offset(handle, param_id)	uint16_t	Returns the byte offset of parameter param_id within the payload image. When precomputed offsets are enabled, this is an O(1) table lookup. Returns 0 if param_id is invalid.
eemul_get_param_size(handle, param_id)	uint16_t	Returns the declared size (in bytes) of parameter param_id, as specified in the descriptor. Returns 0 if param_id is out of range.

9. Troubleshooting

This section describes the most common integration and runtime problems encountered when using EEMUL, their root causes, and how to resolve them.

Table 9-1. Troubleshooting Guide

Symptom	Root Cause	Resolution
eemul_init() returns EEMUL_STATUS_ERR_LAYOUT	The computed block size (header + aligned payload) is larger than EEMUL_FLASH_PAGE_SIZE, or the payload exceeds EEMUL_MAX_PAYLOAD_BYTES. In Sub-Block Mode an oversized block is a fatal error. In Page Mode it is valid if the block fits within the multi-page block boundary.	Either reduce the total parameter payload so it fits within one page alongside the header, increase EEMUL_MAX_PAYLOAD_BYTES to cover the descriptor, switch from compact to full header to understand the exact sizes, or switch to Page Mode and increase EEMUL_NUMBER_OF_FLASH_PAGES to give more space per block.
eemul_init() returns EEMUL_STATUS_ERR_HW	The auto-format failed: erase or program returned an error during the first-boot initialization of the region. Common causes: wrong EEMUL_REGION_END_ADDR or PAGE_SIZE (mismatch with actual hardware), Flash write protection option bytes active on the reserved region, or FMC voltage or timing constraints not met.	Verify EEMUL_REGION_END_ADDR and EEMUL_FLASH_PAGE_SIZE against your device datasheet. Read the option bytes to check sector protection (SPC/WP fields). Verify FMC configuration including wait states, supply voltage, and that fmc_unlock() is called correctly.
Parameters always read back as 0x00 after first boot	First boot with blank Flash. EEMUL auto-formats the region and zero-initializes the first payload (all 0x00). This is expected behavior - the application must write default values after init.	Implement a first-boot detection pattern as shown in Section 7.4. After detecting first boot, use eemul_begin_batch / update / commit_batch to write all factory defaults in a single Flash write.
eemul_write_param() succeeds but data is lost after power cycle	The write was detected as a no-op (differential	On startup, always read the parameter before writing. If the read returns the

Symptom	Root Cause	Resolution
	detection). The shadow buffer contained the same value as src, so no Flash write occurred - but the shadow buffer is not persistent across power cycles. The Flash region may be blank or contain an old value.	freshly-formatted value (0x00), write the default value. Use debugging to verify that the actual bytes written to Flash are correct by reading the block at EEMUL_REGION_START_ADDR with a memory viewer.
eemul_write_param() returns EEMUL_STATUS_ERR_HW with no obvious Flash fault	No safe candidate block is available: all logical blocks in the region are either bad or belong to the current active page (safe rotation prevents erasing the active page), so the commit fails and is reported as EEMUL_STATUS_ERR_HW. This can happen if the region is very small or all blocks have been permanently retired. (The write API never returns EEMUL_STATUS_ERR_NOBLOCK.)	Increase EEMUL_NUMBER_OF_FLASH_PAGES or switch to Sub-Block Mode to increase the number of logical blocks. Check whether blocks were retired erroneously (possibly due to a hardware fault during testing). Note that bad_retry_threshold = 0 is internally rewritten to 1, so the minimum effective threshold is 1 - a value of 0 does not cause immediate retirement.
EEMUL_STATUS_ERR_PARAM from eemul_write_param()	Most commonly: the len argument to eemul_update_param_in_shadow() does not match the size declared for that parameter ID in the descriptor. Less commonly: param_id exceeds param_count, or handle / src pointer is NULL.	Verify that the sizeof() used at the call site matches the sizeof() used in the descriptor for the same type. Using the same typedef (e.g., app_calib_coeff_t) for both the descriptor entry and the write buffer prevents this class of error. Enable assertions to catch mismatches at development time.
Data corrupts after firmware update that changes the descriptor	The stored payload layout in Flash was created by the old firmware descriptor. The new firmware has a different	Implement a descriptor version field as the first parameter (ID 0). After firmware update, read ID 0 and compare against the expected version. If different, write a full set of defaults or

Symptom	Root Cause	Resolution
	number or order of parameters, so the byte offsets are wrong. The old data is read with the new offsets, producing garbage.	migrate the old data to the new layout. Use EEMUL_USE_FULL_BLOCK_HEADER = 1 to enable descriptor CRC detection.

10. Demo Projects

Ready-to-build demo projects ship under demos/ for several GD32 development boards. Each board carries two complementary demos, both with a main.c source, an eemul_config.h tuned for the board, an MCU-specific port file, and project files for the common toolchains (Keil, IAR, and Eclipse/GD32EBuilder). Both demos route their output through EVAL_COM at 115200-8-N-1, so progress can be observed on a serial terminal connected to the board's USB-to-UART port.

10.1. EEMUL_Simple_Demo

The simple demo illustrates the everyday use case: a small parameter store (boot counter, mode, threshold) that survives every power cycle. On the first boot it writes factory defaults through the batch API; on every subsequent boot it reads the current values, prints them, increments the boot counter, and persists the new value. The demo uses the compact header and performs no error injection - it is the recommended starting point when bringing EEMUL up on a new application.

10.2. EEMUL_Functional_Demo

The functional demo exercises EEMUL across a suite of self-checking scenarios: first-boot defaults, persistence across reboot, differential write suppression, wear-leveled rotation, torn-write recovery, and - in full-header builds - payload tamper detection via the block CRC. Each scenario prints step-by-step results and a per-scenario PASS/FAIL line, with a final summary reporting how many scenarios passed. Several scenarios deliberately forge a malformed block into a free Flash slot through the bound port operations to reproduce torn writes and data tampering, then verify that EEMUL recovers safely. The functional demo is suitable for hardware bring-up validation when porting EEMUL to a new MCU.

11. Revision history

Table 11-1. Revision history

Revision No.	Description	Date
1.0	Initial release	Jul.13, 2026

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.