TECHNICAL DOCUMENTATION · COMMUNITY EDITION

Complete technical architecture -- a native FORTH kernel, multi-context model, hardware BIOS and web IDE, for the ESP32-P4.

Version -- living document, 2026-07-08 edition

Written -- by direct inspection of the source code

Translated for -- The Backshed forum, MMBasic / Raspberry Pico community

```
LAYER 4  touchscreen (LVGL) · web IDE (editor, CLI, debugger)
LAYER 3  BIOS (~150 native words) · Mongoose web server (HTTP / WS / MQTT)
LAYER 2  forth_engine.c -- "Model B": web / lcd / background_N contexts
LAYER 1  forth_core.c -- the FORTH virtual machine, 100% portable
```

CONTENTS

Table of Contents

RÉFÉRENCE

Part 0 -- Overview

What is Gami1000, in one sentence?

A small computer (an ESP32-P4 board with a touchscreen) that runs several FORTH programs **in parallel**, each in its own sealed bubble (a "context"), with a built-in web server acting as a code editor, dashboard, and network gateway (WiFi, MQTT).

FORTH is a very old programming language (1970), very different from mainstream languages (Python, C, JavaScript...): instead of writing `result = f(a, b)`, you write `a b f` -- values are pushed onto a "stack" (like a stack of plates) and each word (the FORTH equivalent of a function) takes what it needs from the top of that stack and puts its result back there. This is explained in full, assuming no prior knowledge, in Part 9.

The 4 big layers, bottom to top

console

RULE

Golden rule to remember while reading this document: each layer knows ONLY the layer directly below it. Layer 1 (the FORTH engine) doesn't even know the ESP32 exists -- it runs identically on a Linux PC (the development emulator, `build_emu/`). Layer 3 (BIOS) is what bridges to the real hardware. This strict separation is what lets most of the system be developed/tested on a PC before ever flashing the real board.

Where the vocabulary used in this document comes from

- **FORTH word** = the FORTH equivalent of a function/instruction (e.g. `DUP`, `BIOS-LOG-INFO`). Everything in FORTH is a "word".
- **Dictionary** = the list of all words known at a given moment. Each context has its OWN, completely separate from the others.

- **Context** = an isolated "bubble": its own dictionary, its own task queue, its own system task (see Part 2). There are always at least 2 (`web` and `lcd`), and potentially several more (`background_1` , `background_2` , ...).
- **Stack** (data stack) = where numeric values pile up while waiting to be used by a word. Works like a stack of plates: you push and pop always from the top.
- **BIOS** = here, not a PC's boot BIOS, but the layer of FORTH words that give access to hardware (screen, GPIO, network...) -- vocabulary inherited from the project's predecessor.
- **Role** = the category of a context: `web` , `lcd` , or `background` . Determines which BIOS words are available (e.g. the words that actually draw on the physical screen are visible ONLY in the `lcd` context).

RÉFÉRENCE

Part 1 -- The FORTH core (forth_core.c / forth_core.h)

Role of this layer

This is the language itself: how a word is stored, how it's executed, how numbers are pushed/popped. **Zero ESP32/network dependency** here -- this file compiles and runs identically on the real ESP32 and on a Linux PC (`build_emu/`). This is an absolute project rule: introducing even a single ESP32-specific `#include` here would break the emulator port.

The virtual machine (`forth_vm_t`)

A "virtual machine" here is just a C struct holding ALL the state of a running FORTH interpreter: its stacks, its dictionary, its read position. Each context (web/lcd/background_N) has its OWN, completely independent of the others -- this is what guarantees isolation between contexts.

source.c

```
typedef struct forth_vm_s {
    cell_t  *sp;   // pointer to the top of the data stack
    cell_t  *rp;   // pointer to the top of the return stack
    cell_t  *ip;   // instruction pointer (where execution currently is)
    xt_t     w;    // the word currently executing

    cell_t   ds[256];  // the data stack itself (256 slots)
    cell_t   rs[256];  // the return stack (for nested calls)

    uint8_t *dict;      // the dictionary (all defined words), in PSRAM
    uint8_t *here;      // next free byte in the dictionary
    xt_t     latest;    // the most recently defined word (head of the linked list)

    int      base;      // number base (10 = decimal, 16 = hex)
    int      state;     // 0 = "interpret" mode (execute), 1 = "compile" mode
    // ... + debugger bookkeeping fields, input buffer, etc.
} forth_vm_t;
```

`cell_t` = FORTH's basic unit, "one memory cell". Defined here as `intptr_t` (an integer large enough to hold a memory address) -- 32 bits on the ESP32, 64 bits on a PC. This is the type of EVERY value FORTH manipulates: a number, an address, a boolean (true = -1, false = 0, historical FORTH convention).

PSRAM = extra memory on the ESP32 board (alongside internal RAM, faster but much smaller). The dictionary (128 KB per context) lives in PSRAM because internal RAM is too precious/scarce for that (see Part 2, the event-queue pitfall).

The dictionary -- how a word is stored

The dictionary is a **linked list**: each word knows the address of the word defined right before it (`link`). Looking up a word by name (`FIND` in FORTH) means walking this list backward from the most recently defined word (`vm->latest`) until a name match is found.

```
console

vm->latest --> [most recent word] --link--> [previous word] --link--> ... --> NULL
```

Structure of a word in the dictionary (the "header"):

```
console

+----------+-------+---------+--------------------+--------------+
|  link    | flags | namelen |  name (32 chars max) |  code (XT)    |
| (address | 1 byte| 1 byte  |                    |  = what runs  |
| of the   |       |         |                    |               |
| previous |       |         |                    |               |
| word)    |       |         |                    |               |
+----------+-------+---------+--------------------+--------------+
```

Crucial point for understanding a mechanism used everywhere in this project (see Part 3, "context-agnostic words"): if you define TWO words with exactly the same name in the same dictionary, the second one "shadows" the first -- a name lookup (`FIND`) hits the MOST RECENT one first and stops there, never seeing the older one (which stays in memory but becomes unreachable by name). This is standard FORTH behavior, not a quirk of this project -- but Gami1000 deliberately exploits it so the same word name (e.g. `_DASH-LABEL`) can do different things depending on which context it runs in (see Part 3).

A word's "flags": - `IMMEDIATE` : this word executes EVEN while another word is being compiled (used for `IF` / `THEN` / `DO` / `LOOP` , which must act right away to prepare code, not wait for final execution). - `HIDDEN` : temporarily invisible to lookup -- used for a split second while a word is being defined (between `:` and `;`), to stop it from accidentally calling itself

before it's finished. - `COMPONLY` (compile-only): error if you try to execute this word directly at the keyboard rather than inside a definition -- see the `DO...LOOP` pitfall documented below.

Interpretation vs. compilation -- FORTH's most important distinction

FORTH reads code **one word at a time**, separated by spaces. For each word read, two cases depending on the current state (`vm->state`):

- **Interpret mode** (`state = 0` , the default): the word is immediately looked up in the dictionary and EXECUTED right away. This is what happens when you type a line at the keyboard (REPL).
- **Compile mode** (`state = 1` , enabled by `:` and disabled by `;`): the word is NOT executed -- its address is simply WRITTEN into the dictionary, following the definition currently being built. It will only run later, when someone calls this new word.

script.fth

```
: SQUARE  ( n -- n*n )   DUP * ;
\ While reading this line: ":" enables compile mode.
\ DUP and * are NOT executed here -- their addresses are just written
\ into the dictionary one after another. ";" disables compile mode and
\ ends the definition.

5 SQUARE .   \ THERE, we're in interpret mode: 5 is pushed, SQUARE
             \ is EXECUTED (which in turn executes DUP then *),
             \ "." prints the result: 25
```

PITFALL

A classic pitfall documented in this project: words like `DO` / `LOOP` / `IF` / `THEN` only work if used INSIDE a definition (between `:` and `;`). Typed directly at the keyboard (interpret mode), the loop's code is indeed written into the dictionary (like any compilation), but nothing ever comes along to EXECUTE it -- the line typed at the keyboard is read word by word, sequentially, never "jumping back" the way a real loop would. A script that does `200 0 DO ... LOOP` on its own, without wrapping it in `: word ... ;` , does NOT loop 200 times -- it runs the body exactly once. This is standard FORTH behavior (not a bug in this project's core), but it often surprises newcomers.

The execution loop ("direct-threaded")

Once a COMPOUND word (defined with `:`) is found, how do its components chain together? The core uses a technique called **direct-threaded**: a compound word's "code" is simply a list of addresses of the words that make it up (instead of generated machine code). `forth_inner()` is the loop that walks this list and calls each word one after another:

```
console

Word "SQUARE" compiled in memory:  [address of DUP] [address of *] [address of EXIT]

forth_inner():
  while not done:
      read the next address (vm->ip)
      call the code at that address
      advance vm->ip
```

PITFALL

An important pitfall discovered while building the step-by-step debugger: `forth_inner()` is used ONLY to execute a COMPOUND word (user-defined). A PRIMITIVE word (written directly in C, like `DUP` or a BIOS word) or a plain number typed at the keyboard NEVER goes through `forth_inner()` -- they're executed directly by `forth_outer_eval_line()` (the loop that reads a line typed at the keyboard or in a file). Concrete consequence: a script that never defines or calls a compound word never touches `forth_inner()` at all -- any logic that depended on that function (like the debugger's step mechanism, see Part 8) has to be duplicated in `forth_outer_eval_line()` too.

Inventory of the core's standard words

These words exist in EVERY context, without exception -- they're the foundations of the language itself, not BIOS extensions.

Stack manipulation (move/duplicate/discard values without changing them):

Word	Effect	Explanation
`DUP`	`( n -- n n )`	Duplicates the top value
`DROP`	`( n -- )`	Discards the top value
`SWAP`	`( a b -- b a )`	Swaps the top two values
`OVER`	`( a b -- a b a )`	Copies the 2nd value on top
`ROT`	`( a b c -- b c a )`	Rotates the top 3 values
`-ROT`	`( a b c -- c a b )`	Reverse rotation
`NIP`	`( a b -- b )`	Discards the 2nd value
`TUCK`	`( a b -- b a b )`	Inserts a copy of the top under the 2nd
`2DUP` / `2DROP` / `2SWAP` / `2OVER`	same, but in pairs (handy for string address+length pairs)	
`DEPTH`	`( -- n )`	Number of values currently on the stack
`PICK`	`( ... n -- ... val )`	Copies the value n positions from the top
`?DUP`	`( n -- n n | 0 )`	Duplicates only if non-zero

Arithmetic (integers only -- no floating point, see Part 3 for the Q16.16 convention used to simulate decimals):

Word	Effect
`+` `-` `*` `/` `MOD` `/MOD`	basic operations, `/MOD` returns both quotient and remainder
`NEGATE` `ABS`	sign flip / absolute value
`MAX` `MIN`	larger / smaller of the top two values
`1+` `1-` `2*` `2/`	common shortcuts (+1, -1, x2, /2)

Comparisons (return -1 for true, 0 for false -- FORTH convention): `=` `<>` `<` `>` `<=` `>=` `U<` `U>` (unsigned versions), `0=` `0<` `0>` `0<>` (comparison against zero, faster/more common), `TRUE` `FALSE`.

Memory (read/write directly at an address -- FORTH has no "typed variables" like other languages, just memory addresses):

Word	Effect
`@`	`( addr -- n )` reads a cell at the given address
`!`	`( n addr -- )` writes a cell at the given address
`C@` / `C!`	same, but for a single byte instead of a whole cell
`+!`	`( n addr -- )` adds n to the value already at that address
`HERE`	`( -- addr )` address of the next free byte in the dictionary
`ALLOT`	`( n -- )` reserves n bytes starting at `HERE`
`,`	`( n -- )` writes a cell at `HERE` and advances `HERE`
`MOVE` / `CMOVE`	copies a memory block from one address to another

Return stack (`>R` / `R>` / `R@` , `2>R` / `2R>` / `2R@`): a stack SEPARATE from the data stack, used to temporarily "set aside" a value while the main stack is used for something else, or in the implementation of `DO...LOOP` loops (`I` / `J` give the current loop's index and the enclosing loop's index, respectively).

Output: `EMIT` (prints a character), `TYPE` (prints a string `( c-addr u -- )`), `CR` (newline), `.` (prints a number followed by a space), `.S` (prints the whole stack contents, handy for debugging).

Defining words: - `:` / `;` -- opens/closes a new definition (`: NAME ... ;`). - `VARIABLE NAME` -- creates a word that pushes its OWN memory address (1 reserved cell) -- used with `@` / `!` to read/write into it. - `CONSTANT` -- like `VARIABLE` but the value is fixed at creation and only that value is pushed (not an address). - `CREATE` -- like `VARIABLE` but without reserving a cell -- often followed by `ALLOT` to reserve a buffer of chosen size (e.g. `CREATE BUF 64 ALLOT` reserves 64 bytes named `BUF`).

Control flow (usable ONLY INSIDE a `: ... ;` definition, see the pitfall above): `IF` / `ELSE` / `THEN` (condition), `DO` / `LOOP` / `+LOOP` (bounded loop), `BEGIN` / `UNTIL` , `BEGIN` / `WHILE` / `REPEAT` , `BEGIN` / `AGAIN` (conditional loops).

Character strings: FORTH only knows `(address, length)` pairs for a string (never a terminating character like in C). `S" text"` pushes this pair for a string literal -- **careful, a single shared buffer** is used for this in interpret mode (`str_buf` , a documented pitfall: two `S"` in a row before consuming the first overwrite it, worked around by copying right away into your own buffer via `CREATE` / `MOVE`).

Words specific to this project (not standard ANS FORTH): - `CONTEXTE` / `/CONTEXTE` -- start/end markers for a context file (see Part 2) -- pure no-ops at execution time, their check happens BEFORE evaluation, on the C side. - `BREAKPOINT` -- step-by-step debugger breakpoint (see Part 8). - `WORDS` -- lists every word in the current dictionary. - `DICT-FREE` -- free bytes remaining in the dictionary (128 KB baseline, minus what's already used). - `INCLUDE` / `LOAD-FILE` -- loads and runs a `.fth` file (the second takes the path from the stack `( c-addr u -- )`, the first reads it directly after the word on the same line).

RÉFÉRENCE

Part 2 -- The event engine, "Model B" (forth_engine.c / .h)

Role of this layer

The FORTH core (Part 1) knows how to execute code -- but a system like Gami1000 needs to run SEVERAL FORTH programs at once (one driving the screen, one for the web server, possibly several for background tasks), each reacting to outside events (an MQTT message arrives, a web request arrives, a timer fires...). That's the job of `forth_engine.c`: it turns the "on-demand" FORTH engine (Part 1) into a set of independent FORTH instances, each running continuously in its own system task, listening on its own event queue.

This design is called **"Model B"** in this project -- as opposed to a "Model A" (a single FORTH context fanning out to several outputs) that was tried on the predecessor project and caused a crash that was never resolved -- Model B was chosen as the safer solution, with a much lower integration risk, and has been validated in production ever since.

What a "context" actually is

A context (`forth_ctx_t` , an opaque type -- callers never see its internal fields directly) bundles together:

console

```
+-------------------------------------------------------------+
|  "lcd" context (example)                                    |
|  +---------------------------------------------------------+
|  |  forth_vm_t (Part 1) -- its OWN dictionary, its OWN stacks,   |
|  |  in dedicated PSRAM                                     |
|  +---------------------------------------------------------+
|  +---------------------------------------------------------+
|  |  A DEDICATED FreeRTOS task (ctx_task) -- runs in an endless   |
|  |  loop, waits for an event, handles it, repeats          |
|  +---------------------------------------------------------+
|  +---------------------------------------------------------+
|  |  A dedicated EVENT queue (FreeRTOS queue) -- 32 slots,        |
|  |  internal RAM (not PSRAM)                               |
|  +---------------------------------------------------------+
|  role = FORTH_ROLE_LCD  (determines which BIOS words exist)      |
|  name = "lcd"                                               |
+-------------------------------------------------------------+
```

FreeRTOS (the real-time operating system running underneath all this on the ESP32) lets several "tasks" run at the same time (in reality, alternating very rapidly across the processor's 2 cores) -- each context has its own. This is what guarantees a slow script in the `web` context NEVER blocks the web server itself (which runs in its own task, see Part 5), nor the screen (`lcd` , also its own task).

The 3 roles

source.c

```
typedef enum {
    FORTH_ROLE_WEB,         // exactly 1 instance, always created
    FORTH_ROLE_LCD,         // exactly 1 instance, always created
    FORTH_ROLE_BACKGROUND,  // 0 to N instances, opt-in
} forth_role_t;
```

- **`web`** (always 1 instance): the target of `/api/forth/run` (the IDE's "F5" button) and, broadly, the "general-purpose" context.
- **`lcd`** (always 1 instance): the ONLY context with access to words that actually touch the physical screen (`BIOS-LCD-*` , the real-side `_DASH-*`) -- this filtering is done by CODE, not by convention (a `web` instance that tries to call `BIOS-LCD-CLS` simply gets "unknown word", the word doesn't exist in its dictionary).
- **`background_N`** (0 to N, optional): one instance per `/flash/forth/boot_background_*.fth` file found at boot -- typically for background tasks (polling a sensor in a loop, listening to MQTT) that touch neither the screen nor the web directly.

Creating a context -- why in two steps

source.c

```
forth_ctx_t *forth_ctx_create(const char *name, forth_role_t role);
void         forth_ctx_start(forth_ctx_t *ctx);
```

`forth_ctx_create()` allocates everything (dictionary, stacks, event queue) and registers the base BIOS words -- but does NOT load the boot file and does NOT yet start the task. `forth_ctx_start()` does that next. Why split the two? To let the caller register OTHER additional words in between (for example `forth_state`'s words, Part 4) -- if the boot file were loaded right away inside `create()`, a script using a word not yet registered would fail with "unknown word" at compile time.

The `CONTEXTE` marker -- why and how

Every `boot_<name>.fth` file (automatically loaded when a context starts) MUST begin with `CONTEXTE <name>` and end with `/CONTEXTE`. This is checked **before** evaluating anything (a simple text search in the file, on the C side) -- if the marker is missing, inconsistent (wrong name), or improperly closed, loading is refused with a clear message, rather than loading a file blindly into the wrong context.

Since a product decision made on 2026-07-07, this marker has become the **single source of truth** for deciding where to run ANY `.fth` file (not just boot files): when you press F5 in the web IDE, or type `run myscript.fth` on the command line, the server reads the file's CONTENT, finds `CONTEXTE lcd` (for example) in it, and sends the script to that exact context -- `forth_engine_find_ctx_by_marker()` does this work. Before this decision, F5 always hardcoded the `web` context as its target, which made any script using words reserved for `lcd` fail with "unknown word".

Events -- how a context "reacts" to something

A context spends most of its time doing NOTHING AT ALL, waiting on its event queue (`xQueueReceive`, a call that "sleeps" until something arrives). As soon as an event arrives, its task wakes up, handles it (`dispatch_event()`), then goes back to waiting.

source.c

```
typedef enum {
    FORTH_EV_MQTT,         // an MQTT message arrived
    FORTH_EV_TIMER,        // a scheduled timer fired
    FORTH_EV_UART,         // a frame arrived on the serial port
    FORTH_EV_HTTP,         // an HTTP request dedicated to FORTH
    FORTH_EV_WS,           // a WebSocket message
    FORTH_EV_EVAL,         // code to evaluate, AND wait for the result
    FORTH_EV_GPIO,         // a digital pin changed state
    FORTH_EV_ZIGBEE,       // a Zigbee network event
    FORTH_EV_EVAL_ASYNC,   // code to evaluate, WITHOUT waiting for a reply
    FORTH_EV_RELOAD,       // hot-reload the boot file
    FORTH_EV_RESET,        // fully reset this context
} forth_event_type_t;
```

For every event type (except the last 4, which are internal mechanisms), `dispatch_event()` does the same thing: it fills global FORTH variables with the event's information (the MQTT topic in `CTX-TOPIC`, the message in `CTX-PAYLOAD`, the time in `CTX-TIMESTAMP`...), then calls a FORTH word whose name is fixed by convention -- **if and only if the script has defined it itself**.

console

```
MQTT event arrives
        |
        v
dispatch_event() fills CTX-TOPIC / CTX-PAYLOAD / CTX-TIMESTAMP
        |
        v
calls the word "MQTT-DISPATCH" -- IF the current script defined one
        |                          (otherwise: silently ignored)
        v
: MQTT-DISPATCH  ( -- )                   <- defined BY the user
    CTX-TOPIC 2@  S" temp/living-room" STR-EQ IF
        CTX-PAYLOAD 2@ ... handle the value ...
    THEN
;
```

This mechanism (`*-DISPATCH` words called by naming convention, never a real C-side "handler") is deliberately the ONLY entry point for a script into the event loop -- project rule: **"scripts driven by `dispatch_event()`, no C handlers"**. As of writing, only `MQTT-DISPATCH` is actually wired end to end in the real system (others, like `TIMER-DISPATCH` / `UART-DISPATCH`, exist in the mechanism but nothing on the hardware side triggers those specific events yet -- see the practical manual for the up-to-date list of stubs).

Crucial difference between `FORTH_EV_EVAL` and `FORTH_EV_EVAL_ASYNC` (both "evaluate code", but not the same way): - `FORTH_EV_EVAL` : the caller (e.g. an HTTP request) WAITS for the evaluation to finish and gets the result back. **Must NEVER be used from the web server's task** (`mg_mgr_poll()` , Part 5) -- that would block the ENTIRE server for the whole duration of the script. - `FORTH_EV_EVAL_ASYNC` : fire-and-forget -- the code is posted on the queue, the function returns IMMEDIATELY, the result (or an error) goes out over the serial console / WebSocket channel, never in a direct HTTP reply. This is what `/api/forth/run` (the F5 button) uses -- an absolute project rule, **no exceptions**: this route never blocks the web server, no matter how long the executed script takes.

`CTX-*` variables -- an event's "briefcase"

Variable	Content
`CTX-TOPIC`	MQTT topic / event identifier (address+length pair)
`CTX-PAYLOAD`	message content (address+length pair)
`CTX-TIMESTAMP`	event timestamp
`CTX-DEVICE` / `CTX-ADDR` / `CTX-ENDPOINT`	source identity (Zigbee/UART)
`CTX-VALUE` / `CTX-VALUE-N`	associated (numeric) value
`CTX-ERROR` / `CTX-ERROR-SUB`	error code, if any
`CTX-LQI`	radio link quality (Zigbee)
`CTX-DATA`	free-form description text

These variables are NOT automatically created in the dictionary -- it's up to the script to declare them itself with `VARIABLE` before using them (like any other FORTH variable). They're RELOADED by `dispatch_event()` right before every `*-DISPATCH` call -- so always up to date when the script reads them.

Public API -- the C functions called from outside

Function	Role
`forth_ctx_create(name, role)`	creates a context (dictionary+stacks+queue), executes nothing yet
`forth_ctx_start(ctx)`	loads `boot_<name>.fth` (if present, marker checked), starts the task
`forth_ctx_find(name)`	finds a context already created, by name
`forth_engine_find_ctx_by_marker(buf, ...)`	reads a buffer's `CONTEXTE` marker, finds the target context
`forth_post_event(ctx, type, topic, payload, ...)`	posts an event (with content) onto that context's queue
`forth_post_event_simple(ctx, type, topic)`	same, without content (timers, GPIO)
`forth_post_event_broadcast_role(role, ...)`	posts a COPY of the event to ALL instances of a role (used for MQTT fan-out to every `background_N`)
`forth_eval_sync(ctx, code, result_buf, ...)`	evaluates code and WAITS for the result -- never from the web task, see above
`forth_post_eval_async(ctx, code)`	evaluates fire-and-forget -- what `/api/forth/run` uses
`forth_post_reload(ctx)`	hot-reloads `boot_<name>.fth`, without rebooting the board
`forth_post_kill(ctx)`	fully resets a context (dictionary cleared, boot reloaded)

The event-queue pitfall (internal RAM x N contexts)

Every context has its OWN event queue (`xQueueCreate`), and `xQueueCreate` ALWAYS allocates from internal RAM (never PSRAM, even if asked to) -- internal RAM on the ESP32-P4 is scarce and already under pressure from WiFi (`esp_hosted`). A queue item (`forth_event_t`) contains a `topic[512]` field -- with 32 slots per queue, that's already ~16 KB of internal RAM PER CONTEXT. Before the move to Model B, there was only ONE global queue (that cost was paid just once) -- with several contexts, it's multiplied by the number of contexts. **General rule to remember**: before multiplying a structure originally designed as a singleton, always recompute its memory cost x N. This is precisely why bulky content (the CODE of a whole script to evaluate) never goes through `ev.topic` (capped at 512 bytes) but through `ev.payload` (allocated in PSRAM, free size, via `FORTH_EV_EVAL_ASYNC`).

RÉFÉRENCE

Part 3 -- The BIOS layer: every native FORTH word

Role of this layer

This is the layer that gives the FORTH language (Part 1) access to everything the board actually does: screen, network, files, math. Every "BIOS word" is really a small C function (`forth_bios.c` for the real ESP32, `forth_bios_emu.c` for the Linux emulator) registered into a context's dictionary at startup (`forth_bios_init()` , Part 2).

How a BIOS word is actually written -- always the same shape: a C function taking the VM as a parameter, popping what it needs, doing its job, pushing its result if any:

source.c

```
/* ( pin val -- ) : writes val (0 or 1) to pin */
static void bios_gpio_write(forth_vm_t *vm) {
    cell_t val = forth_core_pop(vm);   /* pop the top (val, arrived last) */
    cell_t pin = forth_core_pop(vm);   /* pop what remained (pin) */
    gpio_set_level((gpio_num_t)pin, val ? 1 : 0);   /* the actual work */
}
```

This function is then associated with a name in a table, walked once at startup to register all the words at once:

source.c

```
static const forth_word_def_t s_bios_words[] = {
    { "BIOS-GPIO-WRITE", bios_gpio_write, 0 },
    /* ... ~150 more entries ... */
};
```

Every category below has TWO implementations: one in `forth_bios.c` (the real ESP32 hardware) and one in `forth_bios_emu.c` (a "fake" equivalent for Linux -- often just a `printf()` showing what WOULD have happened, without real hardware). This lets you test a script's LOGIC on a PC, very fast, before flashing it to the real board.

Filtering by role -- why some words only exist in some places

`forth_bios_init(vm, role)` ALWAYS registers a first group of words (usable by every context, whatever its role), then, ONLY if `role == FORTH_ROLE_LCD`, a second group of additional words that actually touch the physical screen. A `web` or `background_N` instance trying to call a word from that second group gets "unknown word" -- **this isn't a naming convention you could work around, it's structural**: the word was simply never written into ITS dictionary.

```
console
```

```
+-----------------------------------------------------------+
| forth_bios_init(vm, role):                                |
|                                                             |
|   FOR every role:                                          |
|       register s_bios_words[]  (system, GPIO, files,       |
|       math, string, time, MQTT, dashboard preview, JSON...)  |
|                                                             |
|   IF role == FORTH_ROLE_LCD only:                          |
|       register s_bios_words_lcd[]  (BIOS-LCD-*, real       |
|       dashboard _DASH-*)                                    |
+-----------------------------------------------------------+
```

Category: System

Base words, available everywhere.

Word	Effect	Explanation
`BIOS-LOG-INFO` / `-WARN` / `-ERR`	`( c-addr u -- )`	writes a line to the logs (visible on the serial port)
`BIOS-UPTIME-MS`	`( -- ms )`	milliseconds since the board booted
`BIOS-TICKUS`	`( -- us )`	microseconds since boot -- for precisely timing a piece of script
`BIOS-NOOP4`	`( a b c d -- )`	does nothing -- used only to measure the "pure" cost of a FORTH->C call, with no real work
`BIOS-HEAP-FREE`	`( -- bytes )`	free internal RAM right now
`BIOS-PSRAM-FREE`	`( -- bytes )`	free PSRAM right now
`BIOS-RESTART`	`( -- )`	reboots the board immediately
`BIOS-DELAY-MS`	`( ms -- )`	waits ms milliseconds -- **never** inside a `*-DISPATCH` (would block that context for the wait)

Category: Math (simulated fixed-point numbers)

FORTH here only works with INTEGERS (no floating point). To represent a number "with decimals" like 21.5, this project uses the **Q16.16** convention: the real value is multiplied by 65536 before being handled, and divided back by 65536 to display it. `21.5 x 65536 = 1409024` -- this integer "1409024" REPRESENTS 21.5 as long as this convention is followed consistently. The `BIOS-MATH-*` words handle this conversion internally.

Word	Effect
`BIOS-MATH-SIN` / `-COS` / `-TAN`	`( angle-q16 -- value-q16 )` -- angle in RADIANS (not degrees)
`BIOS-MATH-ATAN2`	`( y-q16 x-q16 -- angle-q16 )`
`BIOS-MATH-SQRT`	`( n-q16 -- root-q16 )`
`BIOS-MATH-POW`	`( base-q16 exp-q16 -- result-q16 )`

Example: computing the sine of 90 degrees (pi/2 radians):

script.fth

```
102944 BIOS-MATH-SIN .   \ 102944 = 1.5707963 x 65536 (pi/2 in Q16.16)
                          \ prints roughly 65536 (= 1.0 x 65536, sin(90 deg)=1)
```

Category: String (text manipulation)

FORTH represents a character string as a `(address, length)` PAIR -- never a special terminating character like in C. These words always work in a buffer YOU supply (never a hidden allocation):

Word	Effect
`BIOS-STR-CONCAT`	`( c-addr1 u1 c-addr2 u2 c-addr-dest -- u-total )` glues 2 strings into a supplied buffer
`BIOS-STR-COMPARE`	`( c-addr1 u1 c-addr2 u2 -- n )` compares 2 strings (0 = identical)
`BIOS-STR-FORMAT-INT`	`( n c-addr-dest u-max -- u )` turns an integer into text
`BIOS-STR-FORMAT-FIXED`	`( n-q16 c-addr-dest u-max -- u )` turns a Q16.16 value into readable text ("21.50")

script.fth

```
CREATE BUF 32 ALLOT
1409024 BUF 32 BIOS-STR-FORMAT-FIXED   \ ( u -- ) length written
BUF SWAP TYPE                          \ prints: 21.50
```

Category: Time (clock)

Word	Effect
`BIOS-TIME-NOW`	`( -- epoch )` seconds since Jan 1, 1970
`BIOS-TIME-NTP-SYNC`	`( -- flag )` attempts to sync the clock over the internet (blocks up to 5s)
`BIOS-TIME-DECOMPOSE`	`( epoch -- year month day hour min sec weekday )`
`BIOS-TIME-FORMAT`	`( epoch c-addr-dest u-max -- u )` "YYYY-MM-DD HH:MM:SS" text
`BIOS-TIME-SET-TZ`	`( c-addr u -- )` sets the timezone (e.g. "UTC", "CET-1")

IMPORTANT

Important to know: Gami1000 is a WiFi access point (devices connect to it, but it has no internet connection itself) -- `BIOS-TIME-NTP-SYNC` will therefore almost certainly always fail in this normal mode of operation. Without a successful sync, the clock resets to 1970 on every reboot -- this isn't a bug, it's documented behavior.

Category: GPIO (digital pins)

Word	Effect
`BIOS-GPIO-MODE`	`( pin mode -- )` configures a pin: 0=input, 1=output, 2=input+pull-up, 3=same, pull-down
`BIOS-GPIO-WRITE`	`( pin val -- )` writes 0 or 1 to a pin already set as output
`BIOS-GPIO-READ`	`( pin -- val )` reads a pin's state (-1 for an invalid number)

PITFALL

A hardware pitfall to know about (not a FORTH bug): a pin configured as PURE output cannot, on this chip, read back its own value -- `BIOS-GPIO-READ` on a pin in mode 1 (output) will return garbage, not the value you just wrote.

Category: Files

Word	Effect
`BIOS-FILE-READ`	`( c-addr-path u -- c-addr-buf u | 0 0 )` reads an entire file
`BIOS-FILE-WRITE`	`( c-addr-path u c-addr-data u -- flag )` writes (overwrites)
`BIOS-FILE-APPEND`	same, but appends without overwriting
`BIOS-FILE-EXISTS`	`( c-addr-path u -- flag )`
`BIOS-FILE-DELETE`	`( c-addr-path u -- flag )`
`BIOS-DIR-LIST`	`( c-addr-path u -- c-addr-json u )` lists a folder as JSON

Possible paths: `/flash/...` (internal memory, read-only via the API) or `/sdcard/...` (SD card, read/write -- see Part 7).

Category: Real LCD (`BIOS-LCD-*`) -- `lcd` context only

A "canvas" is an 800x480 image in memory (a large array of colored pixels) drawn on directly -- that's what these words manipulate. The canvas is created automatically (and lazily -- only on the very first draw call, so as not to waste memory if a script never draws) in PSRAM.

Word	Effect
`BIOS-LCD-CLS`	`( -- )` clears the whole screen with the background color
`BIOS-LCD-COLOR`	`( c-addr u -- )` sets the BACKGROUND color (e.g. `S" black"`)
`BIOS-LCD-DRAW-COLOR`	`( c-addr u -- )` sets the current DRAWING color
`BIOS-LCD-COLOR-RGB`	`( r g b -- )` same as DRAW-COLOR but from 3 numbers 0-255 instead of a name
`BIOS-LCD-DRAW-L`	`( x1 y1 x2 y2 -- )` draws a line
`BIOS-LCD-DRAW-C` / `BIOS-LCD-FILL-CIRC`	`( x y r fill|-- )` circle (outline / filled)
`BIOS-LCD-DRAW-R` / `BIOS-LCD-FILL-RECT`	`( x y w h fill|-- )` rectangle (outline / filled)
`BIOS-LCD-DRAW-P`	`( x y -- )` a single pixel
`BIOS-LCD-TEXT`	`( c-addr u x y -- )` text (Montserrat 14 font, current drawing color)
`BIOS-LCD-DRAW-BALL`	`( x y r dir -- )` checkerboard sphere (the historical "Boing Ball" demo)
`BIOS-LCD-BEGIN-FRAME` / `-END-FRAME`	`( -- )` bracket a drawing made of several shapes -- see below
`BIOS-LCD-WIDTH` / `-HEIGHT`	`( -- px )` 800 / 480, so these numbers are never hardcoded
`BIOS-LCD-BACKLIGHT`	`( pct -- )` backlight level (0-100)

Why `BEGIN-FRAME` / `END-FRAME` exist -- to understand this, you need to know that refreshing the physical screen (transferring changed pixels to the panel) costs time proportional to the SIZE of the changed area. If a script draws 5 shapes one by one WITHOUT these brackets, the screen would be refreshed 5 times (5 separate zones, or even the whole screen 5 times) -- with `BEGIN-FRAME` before and `END-FRAME` after, the system accumulates a single "dirty rectangle" (the zone enclosing EVERYTHING that changed) and refreshes the physical screen only ONCE, at the end, over that exact zone. See Part 6 for the details (partial invalidation) and the double-buffer pitfall that follows from it.

Category: "Neutral" drawing (`BIOS-UI-*`) -- available everywhere

Exactly the same vocabulary as `BIOS-LCD-*` (same shape names, same stack signatures), BUT these words NEVER write to the physical screen, even when called from the `lcd` context -- they only send a small JSON message over the network (WebSocket channel),

rendered by the browser on an HTML "simulated screen" canvas (see Part 8). A script written with this vocabulary behaves identically, VISUALLY, whether launched from `web` or from `lcd` -- but draws NOTHING real on the physical panel.

Why this limitation still exists today: making these words draw for real too (in addition to the preview) would have required modifying the already hardware-validated code of the `BIOS-LCD-*` primitives, for a need that wasn't explicitly requested -- an accepted limitation, not an oversight.

Category: Dashboard (`_DASH-*`) -- ready-made widgets

> IMPORTANT
>
> **The single most important mechanism to understand in this whole part**: unlike `BIOS-LCD-*` / `BIOS-UI-*` (two separate vocabularies, with different names), `_DASH-*` words carry the SAME name in every context, but have TWO completely different implementations depending on where they're called:

```
console
```

Example: a small temperature gauge that moves:

script.fth

```
CONTEXTE lcd

_DASH-INIT
VARIABLE GAUGE
50 50 80 0 50 _DASH-ARC GAUGE !     \ gauge 0-50, centered at (50,50), 80 wide

VARIABLE N   0 N !
: LOOP-DEMO  ( -- )
    50 0 DO
        N @ 1+ DUP N !  50 MOD  GAUGE @ SWAP _DASH-ARC-SET
        100 BIOS-DELAY-MS
    LOOP
;
LOOP-DEMO

/CONTEXTE
```

Category: JSON (parsing JSON text)

GENERIC utilities (not tied to a specific mechanism) for reading a value out of JSON text already in memory (e.g. received over MQTT, or stored in `forth_state`, see Part 4). Supported format: "flat", a single level, of the form `{"key":value,"other":value}` -- no lists or nested sub-objects (deliberately out of scope, to stay simple and fast).

Word	Effect
`_JSON-GET-INT`	`( c-addr-json u c-addr-key u -- n )` extracts an integer, 0 if the key is absent
`_JSON-GET-STR`	`( c-addr-json u c-addr-key u -- c-addr u )` extracts a string, empty if absent

script.fth

```
S" {""temp"":2150,""hum"":6000}" ... \ (assume already in memory, see Part 9)
2DUP S" temp" _JSON-GET-INT .  \ prints 2150
```

Category: MQTT / UART / Zigbee / Timers (communication)

Word	Effect
`BIOS-MQTT-PUB`	`( c-addr-topic u c-addr-message u -- )` publishes an MQTT message
`BIOS-MQTT-SUB`	`( c-addr-topic u -- )` subscribes to a topic (assumed no-op -- see below)
`BIOS-UART-SEND`	`( c-addr u port -- )` sends a serial frame -- **stub**, does nothing real currently
`BIOS-ZB-PERMIT` / `-CMD`	Zigbee -- **stub**, logs only
`BIOS-TIMER-START` / `-STOP`	**stub**, creates no real timer -- `TIMER-DISPATCH` currently never fires

`BIOS-MQTT-SUB` "assumed no-op": in this project, ALL received MQTT messages are delivered to ALL `background` contexts (regardless of topic) -- it's up to the FORTH script to check, in its own `MQTT-DISPATCH`, whether `CTX-TOPIC` matches the topic it cares about. `BIOS-MQTT-SUB` exists for form's sake (compatibility with the classic idea of subscribing) but doesn't actually filter anything server-side.

Category: HMI, Config, Device, NVS -- words with varying status

These words exist but have very different degrees of real implementation -- **always check the practical manual before assuming a word has a real hardware effect**:

Word	Real status
`BIOS-HMI-PUSH` / `-PUSH-WEB`	functional -- broadcasts a message to the web (and LVGL for the version without `-WEB`)
`BIOS-DISPLAY-TYPE`	always returns the same fixed value, no real detection
`BIOS-CFG-GET` / `-SET`	always returns "absent"/failure -- not yet wired to real storage
`BIOS-DEV-PROP` / `-SET`	always returns "absent" -- no real Zigbee device behind it
`BIOS-NVS-GET` / `-SET`	functional -- real persistent flash key/value storage (survives reboots)
`BIOS-SDCARD-PRESENT`	functional -- reflects the SD card's real state
`BIOS-NODE-ID`	returns a FIXED identifier ("p4-node"), not unique per board
`BIOS-WORD-EXISTS` / `BIOS-WORD-CALL`	functional -- checks whether a word exists / calls it by name (text) rather than writing it directly into the code

RÉFÉRENCE

Part 4 -- forth_state: state shared between contexts

Why this module exists

Absolute project rule: **two contexts must NEVER write directly into one another** (never share a pointer to another context's VM). So how can a `background_1` context (reading a sensor) let the `lcd` context (which must display it) know its value? Through a SHARED key/value table, protected by a lock (mutex) -- `forth_state.c`. This is the ONLY communication mechanism allowed between two different contexts (outside of Part 2's event system).

```
console
```

The words

Word	Effect
`BIOS-STATE-SET-INT`	`( n c-addr-key u -- )` stores an integer under a key
`BIOS-STATE-GET-INT`	`( c-addr-key u -- n )` reads an integer (0 if absent)
`BIOS-STATE-SET-STR`	`( c-addr-val u c-addr-key u -- )` stores a string
`BIOS-STATE-GET-STR`	`( c-addr-key u -- c-addr u )` reads a string ("" if absent)

Key naming convention

Each producing context writes under a key PREFIXED WITH ITS OWN NAME -- it's the key name itself that says who wrote the data, not a separate field:

```
console
```

```
<context-name>.data       -- the context's full JSON value (recommended)
<context-name>.<field>    -- an individual value (tolerated, simple case)
```

RULE

Boundary rule between JSON and classic stack parameters (so you don't have to re-think this for every new word): - **JSON** when producer and consumer are in DIFFERENT CONTEXTS (e.g. `background_1` -> `lcd`) -- the self-describing format absorbs future added fields without breaking anything already reading it. - **Positional stack parameters** when everything happens WITHIN THE SAME script/context -- there's no distance between the two, JSON wouldn't buy anything here, only cost.

Full example -- `background_1` -> `lcd` communication in JSON

On the `background_1` side (a background context that "invents" a sensor reading every second in this example):

```
script.fth
```

```
CONTEXTE background_1

: TIMER-DISPATCH  ( -- )
    \ build {"temp":2150,"hum":6000,"status":1} by hand --
    \ this core has no automatic number->JSON conversion.
    S" {""temp"":2150,""hum"":6000,""status"":1}" S" bg1.data" BIOS-STATE-SET-STR
;

/CONTEXTE
```

On the `lcd` side (reading this data and displaying the temperature):

```
script.fth
```

```
CONTEXTE lcd

VARIABLE JADDR   VARIABLE JLEN

: READ-TEMP  ( -- temp-q16-or-raw )
    S" bg1.data" BIOS-STATE-GET-STR   JLEN !  JADDR !
    JADDR @ JLEN @  S" temp"  _JSON-GET-INT
;

READ-TEMP .   \ prints 2150
/CONTEXTE
```

WARNING

Point of attention, documented (a choice you must make yourself, not decided by the system): the spec deliberately leaves open the scale factor for decimal numbers in JSON -- Q16.16 (consistent with `BIOS-MATH-*`) OR a more readable factor like x100 (`2150` for 21.50 deg). **Pick ONE factor and document it in the producing script** -- never let the reader guess.

RÉFÉRENCE

Part 5 -- The web server (Mongoose)

Role of this layer

Mongoose is a C library (a ready-made set of functions, pulled from an external project, not written for Gami1000) that runs a COMPLETE web server in a tiny memory footprint -- suited to microcontrollers like the ESP32. `web_server.c` is the part WRITTEN BY this project that uses Mongoose to answer requests.

All of Mongoose runs in **its own FreeRTOS task** (`mg_mgr_poll()` called in a loop) -- **entirely separate** from the FORTH contexts' tasks (Part 2). This is what guarantees a slow FORTH script (even one looping for 10 seconds) never "freezes" the web server: the two run in parallel, never seeing each other directly.

What the web server serves

1. The **UI files** (the web IDE -- HTML/JS/CSS, see Part 8) -- like any classic web server.
2. An **HTTP API** (`/api/...`) -- for listing files, running a FORTH script, logging in, etc.
3. A **WebSocket** channel (`/ws`) -- a permanent connection letting the server PUSH messages to the browser without waiting to be asked (essential for the simulated screen and the real-time console, see Part 8).
4. A **built-in MQTT broker** (port 1883) -- MQTT is a messaging protocol widely used in home automation/IoT ("publishing" a message on a "topic", other devices "subscribing" to that topic). Gami1000 hosts this service itself, no external MQTT server needed.

HTTP route table

Route	Method	Minimum role required	Effect
`/api/auth/login`	POST	none	login (username + password -> token)
`/api/auth/whoami`	GET	none	who am I (based on the supplied token)
`/api/auth/logout`	POST	none	logout
`/api/fs/list`	GET	depends on volume	lists a folder
`/api/fs/read`	GET	depends on volume	reads a file
`/api/fs/write`	POST	depends on volume	writes a file
`/api/fs/delete`	POST	depends on volume	deletes
`/api/fs/mkdir`	POST	depends on volume	creates a folder
`/api/fs/rename`	POST	depends on volume	renames/moves
`/api/version`	GET	none	firmware version
`/api/state`	GET	none	summarized system state
`/api/forth/run`	POST	dev	**runs a `.fth` file** -- the famous "F5"
`/api/forth/eval`	POST	dev	runs ONE line of FORTH code typed at the keyboard (REPL)
`/api/forth/reload`	POST	dev	hot-reloads a context's boot file
`/api/forth/contexts`	GET	dev	lists active contexts + their state (busy/idle)
`/api/forth/pause` / `/kill`	POST	dev	pauses / resets a context
`/api/forth/debug/step` , `/continue` , `/stop`	POST	dev	drives the step-by-step debugger
`/ws`	(upgrade)	none	upgrades the connection to WebSocket
everything else	GET	depends on volume	serves a static file (HTML/JS/CSS)

RULE

An absolute rule never violated in this project: `/api/forth/run` NEVER directly calls the FORTH engine to wait for a result -- it POSTS the event (`forth_post_eval_async` , Part 2) and replies IMMEDIATELY "it's running", regardless of how long the script takes. The target context is determined by READING THE FILE ITSELF (the `CONTEXTE <name>` marker, Part 2) -- never a separate parameter that might not match the file's actual content.

Authentication and permissions (ACL)

3 roles, determined by a text token sent in the `X-Admin-Token` HTTP header (or as a `?_t=` parameter for cases where a header can't be added, like a WebSocket URL):

source.c

```
static const char *get_role(struct mg_http_message *hm) {
    /* reads the sent token (header or _t parameter) */
    if (matches the admin token)  return "admin";
    if (matches the dev token)    return "dev";
    return "guest";               /* no valid token recognized */
}
```

Role	Access to `flash` (internal memory)	Access to `sdcard` (SD card)
`admin`	read (never write -- read-only system area)	everything, any folder
`dev`	invisible	confined to `/sdcard/dev/` only
`guest`	invisible	confined to `/sdcard/guest/` only

The 4 routes that write to the SD card (`write` / `delete` / `mkdir` / `rename`) ALL share the same check function (`sdcard_write_allowed()`) -- a project rule: never reintroduce a different check for one of them (a bug already hit and fixed in the past came exactly from a different, forgotten rule on one of the four).

The WebSocket channel -- how the server "pushes" information

Unlike a classic HTTP request (the browser asks, the server answers, done), a WebSocket connection stays open -- the server can send a message AT ANY TIME, without the browser having asked for anything. This is what lets the simulated screen (Part 8) and console update LIVE while a script is running.

```
console

FORTH word _DASH-BAR-SET ("web" context)
        |
        v
forth_engine_emit_for_vm()  -- builds a small JSON {"op":"fill_rect",...}
        |
        v
web_server_ws_broadcast()  -- puts the message on a queue
        |                     (thread-safe: callable from ANY
        |                     task, including a FORTH task)
        v
mg_mgr_poll() (the Mongoose task) drains the queue, sends over the real
        |                     WebSocket connection
        v
browser receives the message, applyDrawOp() draws it on the HTML canvas
```

PITFALL

A serious pitfall already hit and fixed ("backpressure"): if the FORTH script produces messages faster than the WiFi network can send them, some kind of brake is needed somewhere, otherwise the memory used for pending messages grows without bound until it crashes. Two safeguards exist: (1) pushing onto the queue becomes blocking (instead of silently failing) once it's full -- this naturally slows down the producing FORTH script; (2) the server stops draining that queue as long as the network connection already has too much data waiting to be sent, letting the network catch up before accepting more.

The built-in MQTT broker

An MQTT service runs ON the board itself (port 1883) -- any device on the same WiFi network can publish a message to it, which is immediately relayed to EVERY `background_N` context (see Part 2, `forth_post_event_broadcast_role`) as a `FORTH_EV_MQTT` event, triggering their respective `MQTT-DISPATCH` if defined.

RÉFÉRENCE

Part 6 -- The screen (LVGL, PPA, double framebuffer)

LVGL, in one sentence

LVGL is a C library (like Mongoose, pulled from an external project) specialized in displaying graphical widgets (buttons, bars, text...) on small embedded screens. It handles rendering entirely on its own (drawing a button with rounded corners, its shadow...) -- Gami1000 doesn't reinvent that drawing, it ASKS LVGL to create/modify objects, and LVGL takes care of the rest.

Two coexisting ways of drawing in this project

1. **The manual "canvas"** (`BIOS-LCD-*` , Part 3): a single large LVGL "canvas"-type object (a raw 800x480 image), on which the project's own CODE writes pixels directly (line, circle, text...). Used for freeform drawing (the historical "Boing Ball" demo, for instance).
2. **Real LVGL widgets** (real-side `_DASH-*` , Part 3): standard LVGL objects (button, bar, gauge...) created normally -- LVGL draws and manages them entirely on its own (including their "area to refresh" when they change).

Partial invalidation -- why the screen doesn't fully redraw every time

"Invalidating" an area means marking it as "needs to be redrawn". Redrawing the WHOLE screen on every small change is slow, proportional to the surface area. **Partial invalidation** = redraw ONLY the area that actually changed.

- For the **manual canvas**: every shape drawn grows a "dirty rectangle" (the zone containing everything changed since `BIOS-LCD-BEGIN-FRAME`); `BIOS-LCD-END-FRAME` asks LVGL to refresh ONLY that rectangle. **A fix applied to this mechanism** (a bug found while testing on the real screen, a "ghost" of a bouncing ball's previous position stayed visible): the dirty rectangle from the PREVIOUS frame is also kept in memory, and the UNION of both (current frame + previous frame) is refreshed -- because with 2 alternating physical buffers, the last 2 frames need to be caught up, not just the very latest one, for BOTH buffers to end up correct.
- For **standard LVGL widgets** (`_DASH-*`): LVGL automatically invalidates, on its own, a widget's area when it changes -- but WITHOUT memory of previous frames (unlike the

purpose-built mechanism above). **The same pitfall found again, worse** (a widget that changes often, like a progress bar, appeared to flicker): the solution chosen here, simpler than a per-widget history, is to force a refresh of the WHOLE dashboard PANEL on EVERY widget change, not just the changed widget's area -- with a reasonable number of widgets (a handful of tiles), redrawing the whole panel remains well within the time budget available per frame.

Image rotation (PPA)

This board's physical screen is mounted in PORTRAIT orientation (480 wide x 800 tall), but Gami1000 displays in LANDSCAPE (800 wide x 480 tall) -- so every image must be rotated 90 degrees before being sent to the screen. The PPA (*Pixel Processing Accelerator*, a dedicated piece of circuitry on the ESP32-P4 chip, separate from the main processor) does this rotation in hardware, much faster than a software computation. `lvgl_flush_cb()` (in `app_main.c`) configures and triggers this rotation on every refresh.

The LVGL lock (`lvgl_lock`)

LVGL is NOT designed to be called from several tasks at once without protection. But here, SEVERAL FORTH contexts (each with its own task, Part 2) may want to draw at the same time. `lvgl_lock()` / `lvgl_unlock()` (a "recursive" lock -- the same task can take it several times in a row without blocking itself) protects every LVGL call. **A pitfall already hit**: a NON-recursive lock would deadlock itself (a task trying to re-acquire a lock it already holds would wait forever for itself to release it) -- always use a recursive lock as soon as a call might nest inside another.

The backlight

Controlled by a modulated electrical signal (PWM, via the ESP32's LEDC peripheral) rather than a simple ON/OFF -- allows a gradual 0-100% level (`BIOS-LCD-BACKLIGHT` , Part 3).

RÉFÉRENCE

Part 7 -- Storage (flash LittleFS + SD card)

Two filesystems, two different uses

	`/flash/...`	`/sdcard/...`
Physical medium	Flash memory SOLDERED onto the board	Removable SD card
Filesystem	LittleFS (designed for flash memory)	FAT (universal format, readable on a PC)
Writable via the web API	**No -- read-only**	Yes (depending on role, see Part 5)
Typical content	The web UI itself (HTML/JS/CSS), contexts' boot files	Demo scripts, user data
Who can see what	`admin` only	`admin` (everything), `dev` / `guest` (their own folder only)

Why `/flash` is read-only via the API: this area holds the "system" (the web UI, the boot scripts) -- it's regenerated every time the full firmware is reflashed (the `www.bin` image is built from the project's `flash/` folder, on the development machine, NOT editable from the web IDE).

Mounting the SD card -- a hardware pitfall specific to this board

On THIS specific board (Guition JC4880P443C), the SD card does NOT work with a standard mount -- a particular power rail (LDO, channel 4) that supplies the current needed by the SD card's pins must first be enabled, BEFORE attempting the usual mount. Without this, mounting fails silently (as if no card were inserted, even with one inserted). This detail is specific to this board (documented as a generic pitfall for any similar ESP32-P4 board).

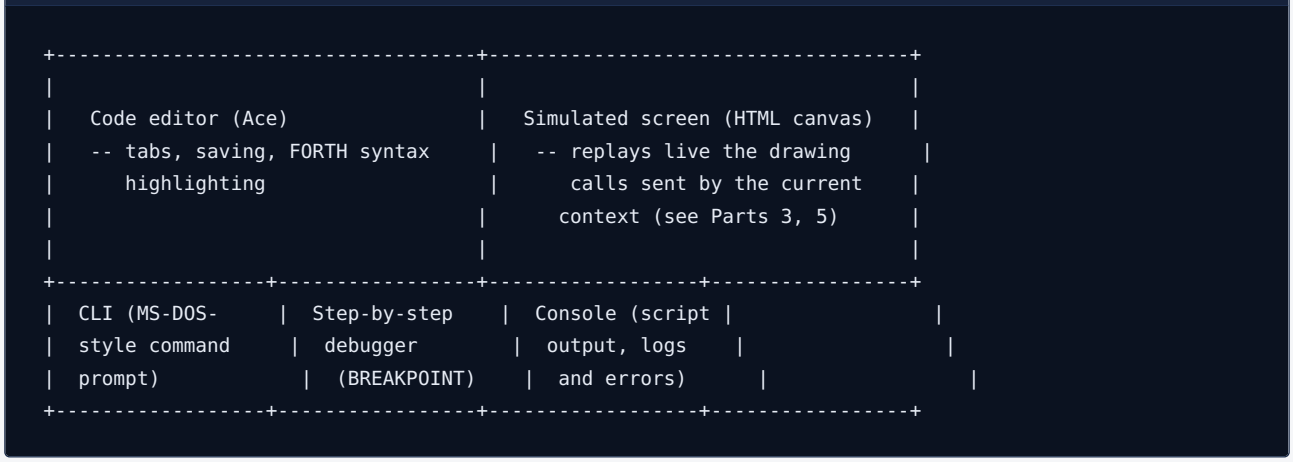
RÉFÉRENCE

Part 8 -- The web IDE (editor, CLI, debugger, simulated screen)

Overview of the screen

The whole interface lives in ONE web page (`filemanager.html`), organized as a grid:

```
console
```



To the left of all this, a sidebar shows the file tree: `flash` , `sdcard` , and a virtual `task` root that lists active FORTH contexts (not real files -- double-clicking a context opens its boot file).

The editor (Ace)

Ace is a ready-made JavaScript library (a full code editor, like the one in VS Code but in the browser) -- `ace.js` / `theme-monokai.js` (the dark visual theme) are loaded as-is. `mode-forth.js` is the ONLY file written specifically for this project: it teaches Ace to recognize FORTH syntax (coloring BIOS words differently from numbers, comments...).

The CLI (MS-DOS-style command line)

A small text command interpreter, inspired by old DOS systems: drive letters (`A:` for flash, `C:` for sdcard), commands `dir` / `cd` / `copy` / `del` / `mkdir` ... and a special `forth` command that switches the CLI into FORTH REPL mode (each line typed is sent as-is to `/api/forth/eval`, run in the currently active editor tab's context).

The step-by-step debugger

The FORTH word `BREAKPOINT` (Part 1), when a script reaches it, halts THAT context's task (other contexts keep running normally) and sends a "snapshot" of the stack contents to the browser. Three buttons drive what happens next: **Step** (run one more word then pause again), **Continue** (resume normally), **Stop** (abort the script).

The simulated screen

A plain HTML `<canvas>` (the browser's native 2D drawing surface, 800x480, same resolution as the physical screen) that replays "draw"/"draw_batch" messages received over WebSocket (Part 5) -- `applyDrawOp()` translates each small JSON message (`{"op":"fill_rect",...}`) into a matching JavaScript drawing call. This mechanism is what lets you see, in the browser, what a script written with `BIOS-UI-*` or `_DASH-*` (preview version) does WITHOUT needing the real physical screen -- handy for fast development/testing.

RÉFÉRENCE

Part 9 -- Programming in FORTH on Gami1000

This chapter is the most important one in this document if you need to WRITE scripts for this system. It starts from zero (how to read FORTH code) and goes up to complete examples by domain.

9.1 -- FORTH for someone who's never seen it

In most languages (Python, C, JavaScript...), you write an operation with the function's name FIRST: `add(3, 4)`. In FORTH, it's the reverse: you write the values first, THEN the word that uses them: `3 4 +`. Every value you type is simply "placed" on a stack (a stack of plates -- you can only add/remove from the top); every word (the equivalent of a function) takes what it needs from the TOP of the stack, does its computation, and places the result back on that same stack.

script.fth

```
3 4 +   \ pushes 3, pushes 4, "+" pops the top 2 values, pushes 7

\ Step by step, what happens on the stack (top is on the right):
\  at the start:       ( empty )
\  after "3":           ( 3 )
\  after "4":           ( 3 4 )
\  after "+":            ( 7 )       <- 3 and 4 are gone, replaced by 7
```

That `( ... -- ... )` notation you'll see all through this document (and in the source code) is called a "stack effect": what comes BEFORE the double dash `--` describes what the word already EXPECTS on the stack (from the oldest value on the left to the most recent/top on the right), after `--` describes what it LEAVES behind.

console

```
( a b -- c )   <- takes 2 values (a below, b on top), leaves 1
```

Why this odd choice? Because FORTH was invented in 1970 for computers with very little memory -- this "stack" notation translates directly into a handful of very simple machine instructions, with no need for a complicated compiler. Today, this choice remains interesting for this project for exactly the SAME reason: an ESP32 doesn't have a PC's power, and a FORTH interpreter is extremely fast compared to other languages that are simple to embed (measured at 300-800x faster than a classic BASIC interpreter on the same hardware).

Defining your own word (the equivalent of writing a function):

script.fth

```
: DOUBLE   ( n -- n*2 )   2 * ;
\ ":" starts the definition, "DOUBLE" is its name, ";" ends it.
\ Everything between the two is what will run EVERY TIME
\ DOUBLE is called.

5 DOUBLE .    \ prints 10
```

Variables (to keep a value "aside", between two calls):

script.fth

```
VARIABLE COUNTER       \ creates a memory slot named COUNTER
0 COUNTER !             \ writes 0 into it ("!" = write)
COUNTER @ .             \ reads it ("@" = read), prints 0
COUNTER @ 1+ COUNTER !  \ read, add 1, write back -- equivalent to "COUNTER++"
```

Character strings: always an (address, length) PAIR, not a single "text object" like in other languages:

script.fth

```
S" hello"    \ pushes TWO values onto the stack: the text's address, and its length (5)
             \ ( -- c-addr u )
S" hello" TYPE   \ TYPE prints a string -- prints: hello
```

9.2 -- The notion of context (why it's THE thing to understand first)

On Gami1000, there is NEVER "a single FORTH program running" -- there are always SEVERAL contexts running AT THE SAME TIME, each in its own completely sealed bubble:

```
console
```

```
+-----------+   +-----------+   +----------------+   +----------------+
|   "web"   |   |   "lcd"   |   | "background_1" |   | "background_2" |
|   always  |   |   always  |   |    optional    |   |    optional    |
|   present |   |   present |   |                |   |                |
+-----------+   +-----------+   +----------------+   +----------------+
      Each has: its OWN word dictionary (words defined in one do NOT
                 exist in the others)
                its OWN data stack
                its OWN system task (they really run in
                parallel, not one after another)
```

IMPORTANT

Important practical consequences: - A word defined (`: MYWORD ... ;`) in the `web` context does NOT exist in the `lcd` context, even though they run on the same board at the same time. - A `VARIABLE` created in one context is NOT visible from another -- to make two contexts communicate, you must go THROUGH `forth_state` (Part 4) or the event system (Part 2), never directly. - Every `.fth` file that serves as a context's starting point MUST declare, right at the top, WHICH context it must run in: `forth CONTEXTE lcd \ ... rest of the script ... /CONTEXTE` This marker is NOT decorative -- it's what the system reads to know where to send the script when you press F5 in the IDE (see Parts 2 and 5). A script without this marker, or with a context name that doesn't exist, is rejected with a clear error message rather than being sent "blindly" somewhere.

- Every context has a **role** that determines which BIOS words are available to it:
- `web` and `background_N` : the "base" words (system, network, files, math...) + the PREVIEW version of `_DASH-*` words (drawing sent to the browser, never to the physical screen).
- `lcd` : the same "base" words + `BIOS-LCD-*` (real drawing on the physical screen) + the REAL version of `_DASH-*` words (real, physically visible widgets).

9.3 -- Examples by domain

System

script.fth

```
CONTEXTE web

S" Script starting" BIOS-LOG-INFO

BIOS-UPTIME-MS .            \ milliseconds since the board booted
BIOS-HEAP-FREE .            \ free internal RAM, in bytes
BIOS-PSRAM-FREE .           \ free PSRAM, in bytes

/CONTEXTE
```

LCD (physical screen -- `lcd` context only)

"Manual" drawing with the canvas (see Part 3 for each word's details):

script.fth

```
CONTEXTE lcd

BIOS-LCD-BEGIN-FRAME            \ start a composite drawing
    S" black" BIOS-LCD-COLOR
    BIOS-LCD-CLS                 \ clears the screen to black
    S" yellow" BIOS-LCD-DRAW-COLOR
    100 100 300 200 -1 BIOS-LCD-DRAW-R   \ yellow rectangle, -1=true=filled in FORTH convention...
                                          \ see Part 3: fill=0 -> outline only, fill<>0 -> filled
    S" Hello Gami1000" 120 150 BIOS-LCD-TEXT
BIOS-LCD-END-FRAME              \ a single physical refresh, for this whole block

/CONTEXTE
```

Or with a ready-made dashboard widget (see Part 3):

script.fth

```
CONTEXTE lcd

_DASH-INIT
S" Hello" 50 50 _DASH-LABEL DROP   \ DROP discards the handle, not needed here

/CONTEXTE
```

Web (preview -- same vocabulary, no access to the physical screen)

The SAME script, almost word for word, but drawing on the browser's SIMULATED screen instead of the physical one:

script.fth

```
CONTEXTE web

BIOS-UI-BEGIN-FRAME
    S" black" BIOS-UI-COLOR
    BIOS-UI-CLS
    S" yellow" BIOS-UI-DRAW-COLOR
    100 100 300 200 -1 BIOS-UI-DRAW-R
    S" Hello (preview)" 120 150 BIOS-UI-TEXT
BIOS-UI-END-FRAME

/CONTEXTE
```

MQTT (receiving a message from any device on the network)

script.fth

```
CONTEXTE background_1

: MQTT-DISPATCH  ( -- )
    \ This word is called AUTOMATICALLY by the system on every MQTT
    \ message received (Part 2) -- CTX-TOPIC/CTX-PAYLOAD are already filled.
    CTX-TOPIC 2@  S" livingroom/temperature"  BIOS-STR-COMPARE 0= IF
        S" Temperature received: " BIOS-LOG-INFO
        CTX-PAYLOAD 2@ BIOS-LOG-INFO
    THEN
;

VARIABLE CTX-TOPIC    2 CELLS ALLOT   \ mandatory declaration, never automatic
VARIABLE CTX-PAYLOAD  2 CELLS ALLOT

/CONTEXTE
```

(to test: publish with `mosquitto_pub -h <board-ip> -t livingroom/temperature -m "21.5"` from another computer on the same network)

Graphics -- simple animation

script.fth

```
CONTEXTE lcd

VARIABLE X   50 X !

: ANIMATE  ( -- )
    100 0 DO
        BIOS-LCD-BEGIN-FRAME
            S" black" BIOS-LCD-COLOR  BIOS-LCD-CLS
            S" cyan"  BIOS-LCD-DRAW-COLOR
            X @ 200 20 BIOS-LCD-FILL-CIRC
        BIOS-LCD-END-FRAME
        X @ 5 + X !
        30 BIOS-DELAY-MS
    LOOP
;
ANIMATE

/CONTEXTE
```

Math (trigonometry, Q16.16)

script.fth

```
CONTEXTE web

: DEG->RAD-Q16  ( degrees -- radians-q16 )
    \ degrees x (pi/180) x 65536, approximated as an integer:
    \ pi/180 x 65536 ~= 1144
    1144 *
;

90 DEG->RAD-Q16 BIOS-MATH-SIN .   \ sine of 90 degrees, roughly 65536 (=1.0)

/CONTEXTE
```

9.4 -- Inter-context communication in JSON, complete end-to-end example

This scenario shows all 3 mechanisms together: a background context `background_1` that "reads a sensor" every 2 seconds (simulated here by a varying value), stores the result as JSON via `forth_state`, and the `lcd` context that reads it and displays it on a `_DASH-*` gauge.

`boot_background_1.fth` :

script.fth

```
CONTEXTE background_1

VARIABLE CTX-TOPIC    2 CELLS ALLOT
VARIABLE CTX-PAYLOAD  2 CELLS ALLOT
VARIABLE N   0 N !

CREATE JBUF 40 ALLOT   \ output buffer -- the assembled JSON
VARIABLE POS            \ current write position in JBUF

\ A pitfall to know: S" ..." has NO escaping mechanism whatsoever -- there
\ is no way to write a literal quote INSIDE an S" ... " string (the first
\ quote encountered always ends the string). Since JSON NEEDS quotes
\ around its keys, they're written one byte at a time (C!, ASCII code 34),
\ and the rest is assembled with CMOVE/BIOS-STR-FORMAT-INT, following a
\ running position that advances with every piece added:
: PUT-QUOTE  ( -- )        34  JBUF POS @ + C!   1 POS +! ;
: PUT-CHAR   ( char -- )        JBUF POS @ + C!   1 POS +! ;
: PUT-STR    ( c-addr u -- )   DUP >R  JBUF POS @ + SWAP CMOVE  R> POS +! ;

: PUBLISH  ( value -- )
    0 POS !
    PUT-QUOTE   S" temp" PUT-STR   PUT-QUOTE
    58 PUT-CHAR                             \ ':' = ASCII code 58
    JBUF POS @ +  20  BIOS-STR-FORMAT-INT  POS +!   \ writes the number, advances POS by its length
    125 PUT-CHAR                             \ '}' = ASCII code 125
    JBUF POS @  S" bg1.data"  BIOS-STATE-SET-STR
;

: TIMER-DISPATCH  ( -- )
    N @ 1+ DUP N !  50 MOD  PUBLISH
;

/CONTEXTE
```

This produces, for instance, `{"temp":27}` -- correctly stored under the `bg1.data` key (naming convention, see above). The code looks verbose for something this simple in another language -- that's the price of the "no embedded JSON library, no hidden allocation" choice made for this core (see Part 3, JSON section): it stays predictable in memory, at the cost of a bit more typing for the developer.

`boot_lcd.fth` (excerpt):

script.fth

```
CONTEXTE lcd

VARIABLE JADDR  VARIABLE JLEN
VARIABLE GAUGE

_DASH-INIT
50 50 100 0 50 _DASH-ARC GAUGE !

: REFRESH  ( -- )
    S" bg1.data" BIOS-STATE-GET-STR  JLEN !  JADDR !
    JADDR @ JLEN @  S" temp"  _JSON-GET-INT
    GAUGE @ SWAP _DASH-ARC-SET
;

: LOOP-DEMO  ( -- )
    BEGIN
        REFRESH
        500 BIOS-DELAY-MS
    AGAIN
;
LOOP-DEMO

/CONTEXTE
```

What this shows: `background_1` knows NOTHING about `lcd` (not its name, not even its existence) -- it just writes to the shared table under a key that carries ITS OWN name (`bg1.data`, Part 4's convention). `lcd`, for its part, just knows WHICH key to read (by construction of the script -- this is never auto-detected). Both contexts run in parallel, each in its own task, never touching each other directly -- exactly the guarantee "Model B" (Part 2) is meant to provide.

9.5 -- What doesn't exist yet (so you don't go looking for it in vain)

- No real touch interaction on `_DASH-*` widgets (checkbox, gauge...) -- they're currently only driven by script, never by a finger on the screen.
- No button, slider, dropdown, or touch text input -- planned for a future separate phase (`_WIDGET-*` vocabulary, see the specs folder).
- `BIOS-TIMER-START` / `-STOP` have no real effect currently (the `TIMER-DISPATCH` word exists in the mechanism but nothing triggers it yet).
- `BIOS-UART-SEND`, `BIOS-ZB-PERMIT` / `-CMD`: do nothing real (no serial/Zigbee link wired up behind these words to date).
- `BIOS-CFG-GET` / `-SET`, `BIOS-DEV-PROP` / `-SET`: always return "absent"/failure.

RÉFÉRENCE

Appendices

Glossary (in the order terms appear in this document)

- **Word**: the FORTH equivalent of a function/instruction.
- **Stack**: a "last in, first out" structure -- like a stack of plates, you only ever touch the top.
- **Push/pop**: place/remove a value from the top of the stack.
- **Dictionary**: the set of words known at a given moment, in ONE specific context.
- **Context**: an independent FORTH instance -- its own dictionary, its own task, its own event queue.
- **Role**: a context's category (`web` / `lcd` / `background`), determines which BIOS words are available.
- **BIOS (in this project)**: the layer of FORTH words giving access to hardware/network/files.
- **Event**: something arriving from the outside (an MQTT message, a web request, a timer...) that wakes up a context so it can react.
- `*-DISPATCH` : the (by-convention) name of a word the user defines to react to a specific event type.
- `CTX-*` : the variables carrying an event's information (topic, content, timestamp...).
- `forth_state` : the shared key/value table letting two different contexts exchange information, without ever touching each other directly.
- **Q16.16**: a convention for representing a decimal number with an integer (real value x 65536).
- **Canvas**: a raw image in memory, drawn on directly, pixel by pixel.
- **Widget**: a ready-made graphical UI element (button, bar, gauge...), generated and managed by LVGL.
- **(Partial) invalidation**: telling LVGL which screen area needs redrawing -- "partial" = only the area that changed, not the whole screen.
- **Framebuffer**: a memory area holding the image actually sent to the screen.
- **PPA**: the ESP32-P4's dedicated circuit that rotates an image (used here to go from the software landscape format to the panel's physical portrait format).
- **Mongoose**: the C library running the web/MQTT server.

- **ACL** (*Access Control List*): the rules determining who is allowed to do what (here: read/write a given folder based on role).
- **WebSocket**: a network connection that stays open, letting the server send messages without waiting to be asked.
- **Ace**: the JavaScript library providing the code editor in the web IDE.

Index -- where to find a specific FORTH word in this document

- Standard language words (DUP, IF, VARIABLE...) -> Part 1
- `CONTEXTE` / `/CONTEXTE` -> Parts 1 and 2
- `BIOS-LOG-*`, `BIOS-UPTIME-MS`, `BIOS-HEAP-FREE`... (system) -> Part 3
- `BIOS-MATH-*` (math) -> Part 3, example in 9.3
- `BIOS-STR-*` (text) -> Part 3
- `BIOS-TIME-*` (clock) -> Part 3
- `BIOS-GPIO-*` -> Part 3
- `BIOS-FILE-*` / `BIOS-DIR-LIST` -> Parts 3 and 7
- `BIOS-LCD-*` (real screen) -> Parts 3, 6, example in 9.3
- `BIOS-UI-*` (preview drawing) -> Part 3, example in 9.3
- `_DASH-*` (dashboard widgets) -> Part 3, examples in Part 3 and 9.4
- `_JSON-GET-*` -> Part 3, examples in Part 4 and 9.4
- `BIOS-MQTT-*` -> Part 3, example in 9.3
- `BIOS-STATE-*` (`forth_state`) -> Part 4, example in 9.4
- `BIOS-HMI-*`, `BIOS-CFG-*`, `BIOS-DEV-*`, `BIOS-NVS-*` -> Part 3 (status table)

Index -- C files by layer

File	Layer	Content
`components/forth_engine/forth_core.c` / `.h`	Part 1	the FORTH engine itself
`components/forth_engine/forth_engine.c` / `.h`	Part 2	contexts, events, Model B
`components/forth_engine/forth_bios.c`	Part 3	BIOS words -- real ESP32
`components/forth_engine/forth_bios_emu.c`	Part 3	BIOS words -- Linux emulator
`components/forth_state/forth_state.c` / `.h`	Part 4	shared key/value table
`components/web_server/web_server.c`	Part 5	HTTP routes, ACL, WebSocket, MQTT
`main/app_main.c`	Part 6	ESP32 startup, LVGL, PPA, backlight
`main/board.h`	Parts 6, 7	this exact board's hardware pinout
`main/main.c`	--	entry point of the Linux EMULATOR only
`flash/www/filemanager.html`	Part 8	the entire web IDE interface
`flash/www/ace/mode-forth.js`	Part 8	FORTH syntax highlighting for Ace

Living document -- expanded over the course of the project. English edition translated for The Backshed forum (MMBasic / Raspberry Pico community), from the original French technical reference, 2026-07-08.