IEScript Lua Automation Manual

Last modified: 2026-07-07

IEScript is the Lua automation layer for Intuition Engine. It is intended for developers who need reproducible emulator automation: boot flows, terminal input, debugger sessions, media playback, screenshots, and recordings.

Scripts use the `.ies` extension.

This manual documents the IE Script Lua API exposed to scripts. It is a developer-facing reference, not a tutorial for Lua itself or for the guest CPU instruction sets.

Contents

Overview

IEScript is designed for deterministic demo automation and tooling:

- drive boot flows and REPL input
- orchestrate debug sessions
- run chip visual/audio transitions
- capture reproducible screenshots and MP4 output

The runtime uses GopherLua (Lua 5.1-compatible semantics).

Getting Started

IEScript is built in - no separate install required. Create a file with the `.ies` extension and pass it on the command line:

```
./bin/IntuitionEngine -script demo.ies
```

A minimal script:

```
sys.print("Hello from IEScript!")
sys.wait_frames(60)  -- wait one second at 60 fps
sys.print("fps:", sys.fps())
```

Launch Modes

CLI file execution

```
./bin/IntuitionEngine -script demo.ies
./bin/IntuitionEngine demo.ies
./bin/IntuitionEngine program.ie64 -script demo.ies
./bin/IntuitionEngine -script demo.ies program.ie64
# With a headless build:
./bin/IntuitionEngine -script render.ies
```

EhBASIC dispatch

From EhBASIC:

```
RUN "demo.ies"
```

ProgramExecutor recognises `.ies` files and routes them to the same script engine used by the `-script` command-line option.

In-window Lua REPL

Press `F8` to toggle the Lua REPL overlay.

Script Runtime Model

Scripts run asynchronously alongside the emulator in a dedicated goroutine. Yield helpers (`sys.wait_frames`, `sys.wait_ms`, visual waits) are frame/timer synchronisation points.

Frame channel

The compositor calls back into the script engine once for every composite pass, including passes where all sources are idle or disabled. This callback sends a notification on an internal channel (capacity 1, non-blocking). When the channel is already full the notification is dropped - this means if script execution between yields takes longer than a frame period, frames are silently skipped rather than queued.

Timing patterns

Pattern	Mechanism	Use case
`sys.wait_frames(n)`	Consumes `n` compositor frame notifications	Frame-based sequencing
`sys.wait_ms(ms)`	Blocks on a wall-clock timer	Time-based delays independent of frame rate
`video.wait_pixel(...)`	Polls each frame until pixel matches	Visual synchronisation

Pattern	Mechanism	Use case
`video.wait_stable(...)`	Polls each frame until hash unchanged	Wait for rendering to settle
`video.wait_condition(...)`	Polls each frame, calls user function	Arbitrary visual predicates

Performance monitoring

Use `sys.frame_time()` to check how many host milliseconds have elapsed since the last yield. If this value consistently exceeds your frame period (e.g. 16 ms at 60 fps), your inter-yield work is too heavy and frames are being dropped.

IEScript can also drive hardware-style demos directly through MMIO. See `sdk/scripts/rotozoomer_ies.ies` for a standalone VideoChip Mode7 rotozoomer that stages a texture with `mem.write_block`, configures blitter registers, and starts MIDI playback through the MIDI MMIO registers.

Important behaviour

- `sys.wait_frames(1)` consumes one compositor frame notification. A notification may already be buffered when the wait begins. The callback still fires when sources are idle, so frame waits continue to advance on a blank display.
- `sys.frame_count()` reports global compositor frame count.
- `sys.frame_time()` reports elapsed host milliseconds since the last yield point.
- All blocking waits (`wait_frames`, `wait_ms`, visual waits) and Lua VM execution respect script cancellation. A cancelled tight loop is interrupted by the VM context and reported as a script error.

Safety and Concurrency Rules

Sandbox

Scripts run in a controlled sandbox by default. Available standard libraries are `base`, `string`, `math`, `table`, `package`, the IEScript-provided `bit32`, and a restricted `os` table. The `io` and `debug` libraries are not opened.

Removed globals: `dofile`, `loadfile`, `load`, `loadstring`, `module`, and `collectgarbage`.

Removed `os` functions: `execute`, `exit`, `remove`, `rename`, `tmpname`, and `setlocale`. Kept `os` functions include `time`, `date`, `clock`, `getenv`, and `difftime`.

`require` only loads Lua modules from approved script roots: the current script directory and `sdk/scripts/`. Module names containing `..` or path separators are rejected. Native modules and `package.loadlib` are not available.

Scripts that depend on `io.open`, `os.execute`, arbitrary host paths, native Lua modules, or dynamic code loading are outside this sandboxed API and must be rewritten before they can be used as normal end-user IEScript files.

Raw RAM access

Raw RAM access requires freezing:

- `mem.read*`, `mem.write*`, `mem.read_block`, `mem.write_block`, `mem.fill`

Use:

```
cpu.freeze()
-- raw memory operations
cpu.resume()
```

MMIO access

MMIO access is allowed without freezing. Block and aggregate operations validate the full byte range. A range that starts in MMIO but spills into RAM is rejected unless the script has frozen the CPU.

Freeze reference counting

Freeze requests are reference-counted across API surfaces (`cpu.*`, `dbg.*`). One subsystem closing a freeze source does not implicitly clear another active freeze source. Specifically:

- `cpu.freeze()` increments the global freeze counter.
- `cpu.resume()` decrements it (floor of 0 - extra resumes are harmless).
- `dbg.open()` / `dbg.freeze()` activate the monitor and add one script-owned freeze on the first nested open.
- `dbg.close()` / `dbg.resume()` release that script-owned freeze when the nested open count reaches zero.

If a script errors or is cancelled, script-owned CPU freezes, debugger opens, and freezes made through `audio.freeze()` are automatically released. Audio is restored to the state it had when the script started only if the script touched audio through the `audio.freeze()` / `audio.resume()` API.

Host file paths

Host path access is restricted to approved roots. Relative reads search the current script directory first, then `sdk/scripts/`. The `sys.read_file` and `sys.copy_file` source path helpers also allow repository diagnostic assets under `sdk/examples/` and `sdk/ab3d64/`. Absolute read paths are allowed only when the resolved target is still inside an approved root. Writes are script-relative only and cannot use absolute paths. Parent directories for writes must already exist unless the script creates them first with `sys.mkdir`. Existing output files and symlinks are resolved before the root check, so an output symlink cannot point outside the script root. Special `cpu.load("EMUTOS")` and `cpu.load("AROS")` tokens are unchanged.

The policy applies to `cpu.load`, audio loaders, recording and screenshots, output capture, debugger state/memory files, debugger trace files, monitor scripts launched through `dbg.run_script`, the Lua `media.load` helpers, and the `sys.*_file` helpers. The lower media MMIO loader also treats its configured media base directory as an approved root: absolute paths are accepted only when their resolved target stays under that root, and traversal or symlink escapes are rejected.

`sys.mkdir(path)` is the only script API that creates output directories. It accepts a script-relative directory path, creates missing components one at a time, and rejects absolute paths, `..` traversal, non-directory components, and symlinked directory components. This prevents a script from creating directories outside its write root through a symlink such as `out -> /tmp/outside`.

`sys.emutos_drive(path [, drive])` is a host-directory mapping control, not a script asset loader. It passes `path` to the EmuTOS GEMDOS drive configuration for the next EmuTOS boot or mode switch and does not resolve it through the script-relative read/write policy above.

Module Reference

Module Reference Conventions

Each API entry is written as a Lua call signature followed by its side effects and return contract.

Notation	Meaning
`name(arg)`	Required argument

Notation	Meaning
`name([arg])`	Optional argument
`name(arg [, optional])`	Required argument followed by an optional argument
`Returns: nothing`	The function leaves no Lua value on the stack
`Returns: value`	The function pushes the named value or table
`Raises on ...`	The binding calls the Lua error path and aborts the current script unless caught by Lua code

Path-taking functions use the script sandbox rules from section 5 unless the entry explicitly says that the path is a host mapping control. Raw RAM reads and writes require the CPU freeze contract described in section 5. MMIO helpers are separate from raw RAM helpers: they go through the bus-visible I/O path and may trigger device side effects.

Error and Result Rules

Bindings reject malformed arguments through the Lua error path. Unless the entry documents a returned status table or boolean, an invalid argument, missing file, rejected path, unsupported selected CPU, failed monitor command, or out-of-range device field aborts the current script and is reported as the script's last error.

Return values use Lua types directly:

Contract	Lua result
`Returns: nothing`	No value is pushed; assigning the call result yields `nil`
`Returns: boolean`	`true` or `false`
`Returns: number`	Lua number, usually converted from an unsigned guest or host value
`Returns: string`	UTF-8 host string or byte string as described by the entry
`Returns: table`	Plain Lua table with field names documented in the entry

Monitor integration has an additional filter. `dbg.command` and `dbg.command_output` reject monitor commands that read or write arbitrary host files or mutate monitor script state: `save`, `load`, `ss`, `sl`, `script`, `macro`, and `trace file`; `trace file off` is allowed. Use the specific IEScript file, snapshot, symbol, trace, and debugger helpers instead, because those helpers apply the script path policy described earlier.

`sys`

Timing, diagnostics, lifecycle.

`sys.wait_frames(n)` - Consume `n` compositor frame notifications from the frame channel. A pending notification may already be buffered when the call starts. Returns: nothing.

`sys.wait_ms(ms)` - Block for `ms` milliseconds (wall-clock timer). Returns: nothing.

`sys.print(...)` - Print all arguments to host stdout, space-separated, with trailing newline. Returns: nothing.

`sys.log(...)` - Log all arguments to host stdout (mirrors `sys.print` currently). Returns: nothing.

`sys.time_ms()` - Current Unix time in milliseconds. Returns: number.

`sys.frame_count()` - Global compositor frame count since engine start. Returns: number.

`sys.frame_time()` - Milliseconds elapsed since the last yield point (wait_frames, wait_ms, or visual wait). Useful for detecting slow scripts. Returns: number.

`sys.fps()` - Current output backend refresh rate in Hz. The compositor tick used for frame callbacks remains 60 Hz. Returns: number.

`sys.perf_report()` - `sys.perf_report()` returns a string subsystem performance report; it is empty when `IE_PERF_ACCT` is off or no subsystem counters have recorded work. The report contains one line per non-empty bucket in the form `name: total-ms ops avg-us/op`. Returns: string.

`sys.perf_reset()` - `sys.perf_reset()` resets subsystem performance counters and returns nothing. Use it at the start of a measurement window before calling `sys.perf_report()`. Returns: nothing.

`sys.quit()` - Stop any active recording and shut down the emulator. Returns: nothing.

`sys.exit([code])` - Stop any active recording and exit the engine with optional integer `code` (default 0). Unlike `sys.quit`, this propagates an exit status to the host process via the engine's exit hook. Returns: nothing.

`sys.emutos_drive(path [, drive])` - Set a GEMDOS drive host directory for the next EmuTOS boot. The optional `drive` parameter is the drive letter index (default 20 = drive U:; A:=0, C:=2, etc.). Updates both the program executor and the internal boot path so the next `EMUTOS` command (from BASIC) or EmuTOS mode switch will map the chosen drive to the specified directory. Returns: nothing.

`sys.mkdir(path)` - Create a script-relative directory path and any missing parents. Returns the validated absolute host path that was created or already existed. Raises on absolute paths, `..` traversal, symlinked directory components, non-directory components, or filesystem errors.

`sys.read_file(path)` - Read a file from an approved read root and return its contents as a Lua string. Relative paths search the current script directory and `sdk/scripts/`; this helper also permits repository diagnostic assets under `sdk/examples/` and `sdk/ab3d64/`. Raises on path validation or read errors.

`sys.write_file(path, data)` - Write Lua string `data` to script-relative `path`. Parent directories must already exist; call `sys.mkdir` first when needed. Raises on absolute paths, traversal, symlink escapes, missing parents, or write errors. Returns: nothing.

`sys.copy_file(src, dst)` - Copy file contents from approved read path `src` to script-relative write path `dst`. Source path rules match `sys.read_file`; destination path rules match `sys.write_file`. Returns the validated absolute destination path. Raises on path validation, missing parents, read, or write errors.

`sys.capture_output(path)` - Redirect host stdout/stderr to script-relative `path`. Returns: nothing. Raises on path validation or file errors.

`sys.capture_output_off()` - Stop output capture and restore host stdout/stderr. Returns: nothing.

Example:

```
sys.print("fps", sys.fps())
sys.wait_frames(120)
sys.print("frame_time:", sys.frame_time(), "ms")
```

---

cpu

CPU lifecycle and mode.

`cpu.load(path)` - Load a program binary from `path` into the active CPU. The file format must match the active CPU mode. The special values `"EMUTOS"` and `"AROS"` trigger OS boot without a file path (ROM resolved via CLI flags or embedded image). Returns: nothing. Raises on error.

`cpu.load_stopped(path)` - Load an IE64 or IE32 program from an approved read path, reset that CPU, and leave it stopped. This is intended for scripts that need to set registers, memory, breakpoints, or JIT state before starting execution. It is supported only when the active CPU is IE64 or IE32; it does not accept the `"EMUTOS"` or `"AROS"` sentinel tokens. Returns: nothing. Raises on path, read, or unsupported-CPU errors.

`cpu.reset()` - Perform a hard reset of the emulator (all CPUs and devices). Returns: nothing. Raises on error.

`cpu.freeze()` - Increment the global freeze counter, pausing CPU execution for safe memory access. Returns: nothing.

`cpu.resume()` - Decrement the global freeze counter. Execution resumes when the counter reaches zero. Extra resume calls beyond zero are harmless (counter floors at 0). Returns: nothing.

`cpu.start()` - Start execution on the active CPU. Returns: nothing.

`cpu.stop()` - Stop execution on the active CPU. Returns: nothing.

`cpu.is_running()` - Check whether the active CPU is currently executing. Returns: boolean.

`cpu.mode()` - Return the active CPU type as a string. Returns: one of `"ie32"`, `"ie64"`, `"m68k"`, `"z80"`, `"x86"`, `"6502"`, or `"none"`.

`cpu.jit_enabled()` - Check whether JIT compilation is currently enabled for the active CPU. Supported for m68k, z80, x86, 6502, and ie64 when the current host build includes that CPU's JIT backend; returns `false` for any other CPU or unavailable backend. Returns: boolean.

`cpu.set_jit_enabled(enabled)` - Enable or disable JIT for the active CPU. Raises if the CPU is currently running, if the platform build does not provide a JIT for that CPU, or if the selected CPU does not support script-controlled JIT (currently m68k, z80, x86, 6502, and ie64). x86, m68k, z80, and 6502 JIT backends are amd64 host paths; IE64 also has arm64 host paths as described in architecture.md. On a successful disable the JIT is turned off immediately. Returns: nothing.

`cpu.execution_mode()` - Report the effective execution mode for the active CPU. Returns: `"jit"` if a JIT is enabled and available for that CPU, otherwise `"interpreter"`.

`cpu.jit_stats()` - Return JIT diagnostic counters for the active CPU. In m68k mode, returns a table with `instruction_count`, `native_blocks`, `last_native_pc`, `fallback_instructions`, `bailouts`, `last_fallback_pc`, `last_fallback_opcode`, `fallback_opcodes`, `native_pcs`, `native_pc_ring`, and `compile_failures`. `fallback_opcodes` contains up to 16 entries with `opcode`, `count`, and `pc`; `native_pcs` contains up to 16 entries with `pc` and `count`; `compile_failures` contains up to 16 entries with `pc`, `count`, and `error`; `native_pc_ring` contains recent native-block PCs. Other CPU modes return an empty table. Returns: table.

Example:

```
cpu.load("program.ie32")
cpu.start()
sys.wait_frames(60)
sys.print("CPU mode:", cpu.mode(), "running:", cpu.is_running())
cpu.freeze()
-- safe to read/write memory here
cpu.resume()
```

---

mem

Memory/bus operations. All `mem.*` functions require `cpu.freeze()` for raw RAM addresses. MMIO addresses (device registers) are allowed without freezing.

The `mem.*` module is a raw 32-bit shared-bus API. Each address argument is converted to `uint32` before access and the functions call the shared bus directly, not the focussed CPU adapter. Addresses above `0xFFFFFFFF` are truncated to their low 32 bits. This module is for bus-visible 32-bit physical RAM/MMIO ranges; it is not an above-4GiB IE64 RAM or CPU-virtual-address API.

`mem.read8(addr)` - Read one byte from bus address `uint32(addr)`. Returns: number (0..255).

`mem.read16(addr)` - Read a 16-bit word from bus address `uint32(addr)`. Returns: number.

`mem.read32(addr)` - Read a 32-bit word from bus address `uint32(addr)`. Returns: number.

`mem.write8(addr, value)` - Write one byte `value` to bus address `uint32(addr)`. Returns: nothing.

`mem.write16(addr, value)` - Write a 16-bit word `value` to bus address `uint32(addr)`. Returns: nothing.

`mem.write32(addr, value)` - Write a 32-bit word `value` to bus address `uint32(addr)`. Returns: nothing.

`mem.read_block(addr, len)` - Read `len` bytes starting at bus address `uint32(addr)`. Returns: string (raw bytes).

`mem.write_block(addr, bytes)` - Write the raw byte string `bytes` starting at bus address `uint32(addr)`. Returns: nothing.

`mem.fill(addr, len, value)` - Fill `len` bytes starting at bus address `uint32(addr)` with byte `value`. Returns: nothing.

Detailed `mem.*` contracts:

- `mem.read8(addr)` returns number and truncates addr to uint32
- `mem.read16(addr)` returns number and truncates addr to uint32
- `mem.read32(addr)` returns number and truncates addr to uint32
- `mem.write8(addr, value)` returns nothing and truncates addr to uint32
- `mem.write16(addr, value)` returns nothing and truncates addr to uint32
- `mem.write32(addr, value)` returns nothing and truncates addr to uint32
- `mem.read_block(addr, len)` returns a raw byte string; len must be >= 0
- `mem.write_block(addr, bytes)` writes a raw byte string and returns nothing
- `mem.fill(addr, len, value)` fills bytes, returns nothing, and requires len >= 0

Example:

```
cpu.freeze()
local val = mem.read32(0x1000)
sys.print("value at 0x1000:", val)
mem.write8(0x2000, 0xFF)
mem.fill(0x3000, 256, 0)
local block = mem.read_block(0x1000, 16)
cpu.resume()
```

---

term

Terminal automation for driving the emulated terminal I/O.

`term.type(str)` - Enqueue each byte of `str` as keyboard input to the terminal. Does not append a newline. Returns: nothing.

`term.type_line(str)` - Enqueue `str` followed by a newline character. Returns: nothing.

`term.read()` - Drain and return all pending terminal output. Returns: string.

`term.clear()` - Drain and discard all pending terminal output. Returns: nothing.

`term.echo(on)` - Enable or disable terminal echo. Returns: nothing.

`term.wait_output(pattern, timeout_ms)` - Poll terminal output every 10 ms until `pattern` (a plain string, not a regex) is found or `timeout_ms` expires. Accumulates output across polls. Returns: boolean (`true` if pattern found, `false` on timeout).

`term.mouse_move(x, y)` - Set the mouse position. Coordinates are clamped to the compositor frame bounds (negative values become 0, values beyond frame dimensions are clamped to the edge). Returns: nothing.

`term.mouse_delta(dx, dy [, button])` - Add signed relative mouse deltas to `MOUSE_DX` and `MOUSE_DY`. The optional `button` value uses the same encoding as `mouse_click`: 1 = left, 2 = right, 3 = both. Returns: nothing.

In desktop builds, captured relative mouse mode can be released manually with `Ctrl+Alt`; left-click inside the IE window recaptures while the guest still has `MOUSE_CTRL` bit 0 set. This host escape does not change guest-visible `MOUSE_CTRL` and does not affect scripted `term.mouse_delta` injection.

`term.mouse_release()` - Clear the script-side mouse-override flag set implicitly by `term.mouse_move`, `term.mouse_delta`, `term.mouse_click`, and `term.mouse_double_click`. Call this when a script is done driving the mouse so the host's normal mouse policy resumes. Returns: nothing.

`term.mouse_click(x, y [, button])` - Perform a single mouse click at (x, y). Coordinates are clamped to frame bounds. The optional `button` parameter specifies which button: 1 = left (default), 2 = right, 3 = both. The implementation moves the pointer, waits 50 ms, holds the button for 60 ms, releases it, then waits 50 ms before returning. Returns: nothing.

`term.mouse_press(x, y [, button])` - Move the mouse to clamped coordinates, wait 50 ms, press the selected button, wait another 50 ms, and leave the button pressed. Use `term.mouse_release()` to clear the button state and return to normal host mouse handling. The optional `button` parameter uses the same encoding as `mouse_click`: 1 = left, 2 = right, 3 = both. Returns: nothing.

`term.mouse_double_click(x, y [, button])` - Perform a double click at (x, y). Coordinates are clamped, button values are the same as `mouse_click`. The implementation moves the pointer, waits 50 ms, then performs two 60 ms button holds separated by 80 ms gaps. Returns: nothing.

`term.scancode(code)` - Inject a raw Atari ST scancode (make or break) into the scancode ring buffer. Code must be 0..255. Returns: nothing.

`term.key_press(code [, hold_ms])` - Inject a key press: enqueues the make code, waits `hold_ms` milliseconds (default 50), then enqueues the break code (make code OR 0x80). Code must be 0..127 (make codes only). Returns: nothing.

Example:

```
term.type_line('PRINT "HELLO"')
local ok = term.wait_output("HELLO", 2000)
if not ok then
  error("expected output not seen")
end
```

Mouse and keyboard example (EmuTOS GEM automation):

```
-- Move mouse to menu bar and click
term.mouse_click(320, 5)
sys.wait_ms(200)

-- Double-click a drive icon
term.mouse_double_click(600, 60)
sys.wait_ms(1000)

-- Press Enter key (Atari ST scancode 0x1C)
term.key_press(keys.ENTER)
```

---

keys

Global constant table of Atari ST scancodes for use with `term.scancode()` and `term.key_press()`. All values are make codes (0x00..0x7F).

Constant	Value	Constant	Value
`keys.ESCAPE`	0x01	`keys.BACKSPACE`	0x0E
`keys.TAB`	0x0F	`keys.ENTER`	0x1C
`keys.SPACE`	0x39	`keys.LSHIFT`	0x2A
`keys.RSHIFT`	0x36	`keys.LCTRL`	0x1D
`keys.CAPSLOCK`	0x3A	`keys.F1`	0x3B
`keys.F2`	0x3C	`keys.F3`	0x3D
`keys.F4`	0x3E	`keys.F5`	0x3F
`keys.F6`	0x40	`keys.F7`	0x41
`keys.F8`	0x42	`keys.F9`	0x43
`keys.F10`	0x44	`keys.UP`	0x48
`keys.DOWN`	0x50	`keys.LEFT`	0x4B
`keys.RIGHT`	0x4D	`keys.A`	0x1E
`keys.B`	0x30	`keys.C`	0x2E
`keys.D`	0x20	`keys.E`	0x12
`keys.F`	0x21	`keys.G`	0x22
`keys.H`	0x23	`keys.I`	0x17
`keys.J`	0x24	`keys.K`	0x25
`keys.L`	0x26	`keys.M`	0x32
`keys.N`	0x31	`keys.O`	0x18
`keys.P`	0x19	`keys.Q`	0x10
`keys.R`	0x13	`keys.S`	0x1F
`keys.T`	0x14	`keys.U`	0x16

Constant	Value	Constant	Value
`keys.V`	0x2F	`keys.W`	0x11
`keys.X`	0x2D	`keys.Y`	0x15
`keys.Z`	0x2C	`keys.DIGIT_1`	0x02
`keys.DIGIT_2`	0x03	`keys.DIGIT_3`	0x04
`keys.DIGIT_4`	0x05	`keys.DIGIT_5`	0x06
`keys.DIGIT_6`	0x07	`keys.DIGIT_7`	0x08
`keys.DIGIT_8`	0x09	`keys.DIGIT_9`	0x0A
`keys.DIGIT_0`	0x0B	`keys.MINUS`	0x0C
`keys.EQUAL`	0x0D		

`audio`

Sound chip and player controls.

Core

`audio.start()` - Start the sound chip. Returns: nothing.

`audio.stop()` - Stop the sound chip. Returns: nothing.

`audio.reset()` - Reset the sound chip to initial state. Returns: nothing.

`audio.freeze()` - Freeze audio generation (silence output). Returns: nothing.

`audio.resume()` - Resume audio generation after a freeze. Returns: nothing.

`audio.write_reg(addr, value)` - Write a 32-bit `value` to sound chip register at bus address `addr`. This is an MMIO write (no freeze required). Returns: nothing.

Master Mix and Dynamics

These functions tune the sound chip's master output stage (post-mix gain, auto-leveller, and compressor). All operate on the live sound chip and take effect immediately; no freeze required.

`audio.set_master_gain_db(db)` - Set master output gain in decibels (float). Returns: nothing.

`audio.get_master_gain_db()` - Read the current master gain in decibels. Returns: number.

`audio.set_master_auto_level_enabled(on)` - Enable or disable the master auto-leveller. Returns: nothing.

`audio.configure_master_auto_level(target_db, min_gain_db, max_gain_db, attack_ms, release_ms)` - Configure the auto-leveller with five floating-point parameters: target loudness in dB, minimum and maximum allowed gain in dB, attack and release time constants in ms. Argument order matches `SoundChip.ConfigureMasterAutoLevel`. Returns: nothing.

`audio.set_master_compressor_enabled(on)` - Enable or disable the master compressor. Returns: nothing.

`audio.configure_master_compressor(threshold_db, ratio, attack_ms, release_ms, knee_db, makeup_db, lookahead_ms)` - Configure the compressor with seven floating-point parameters. Returns: nothing.

`audio.use_showreel_normalizer_preset()` - Apply the canonical "showreel" preset (auto-level + compressor settings tuned for the demo showreel). Returns: nothing.

`audio.reset_master_dynamics()` - Reset auto-level and compressor internal state (gain envelopes, look-ahead buffer). Does not change the configured parameters. Returns: nothing.

PSG (AY-3-8910 / YM2149)

`audio.psg_load(path)` - Load a PSG music file from an approved read path. Supported extensions: `.vgm`, `.vgz`, `.vtx`, `.vt`, `.ym`, `.ay`, `.snd`, `.sndh`, `.pt3`, `.pt2`, `.pt1`, `.stc`, `.sqt`, `.asc`, `.ftc`. Returns: nothing. Raises on error.

`audio.psg_play()` - Start PSG playback. Returns: nothing.

`audio.psg_stop()` - Stop PSG playback. Returns: nothing.

`audio.psg_is_playing()` - Check whether the PSG engine is currently playing. Returns: boolean.

`audio.psg_metadata()` - Return metadata for the currently loaded PSG file. Returns: table with fields:

Field	Type	Description
`title`	string	Track title
`author`	string	Composer name
`system`	string	Target system

SID (MOS 6581/8580)

`audio.sid_load(path [, subsong])` - Load a SID music file from an approved read path. The optional `subsong` parameter selects a sub-song index (default 0). Returns: nothing. Raises on error.

`audio.sid_play()` - Start SID playback. Returns: nothing.

`audio.sid_stop()` - Stop SID playback. Returns: nothing.

`audio.sid_is_playing()` - Check whether the SID player is currently playing. Returns: boolean.

`audio.sid_metadata()` - Return metadata for the currently loaded SID file. Returns: table with fields:

Field	Type	Description
`title`	string	Track title
`author`	string	Composer name
`released`	string	Release information
`duration`	string	Duration text

TED (MOS 7360/8360)

`audio.ted_load(path)` - Load a TED music file from an approved read path. Returns: nothing. Raises on error.

`audio.ted_play()` - Start TED playback. Returns: nothing.

`audio.ted_stop()` - Stop TED playback. Returns: nothing.

`audio.ted_is_playing()` - Check whether the TED player is currently playing. Returns: boolean.

POKEY (Atari C012294)

`audio.pokey_load(path)` - Load a POKEY music file (SAP) from an approved read path. Returns: nothing. Raises on error.

`audio.pokey_play()` - Start POKEY playback. Returns: nothing.

`audio.pokey_stop()` - Stop POKEY playback. Returns: nothing.

`audio.pokey_is_playing()` - Check whether the POKEY player is currently playing. Returns: boolean.

AHX (Abyss' Highest eXperience)

`audio.ahx_load(path)` - Load an AHX music file from an approved read path. Returns: nothing. Raises on error.

`audio.ahx_play()` - Start AHX playback. Returns: nothing.

`audio.ahx_stop()` - Stop AHX playback. Returns: nothing.

`audio.ahx_is_playing()` - Check whether the AHX engine is currently playing. Returns: boolean.

MIDI/MUS

`audio.midi_load(path)` - Load an SMF `.mid`/`.midi` or Doom `.mus` file from an approved read path. Returns: nothing. Raises on error.

`audio.midi_play()` - Start MIDI playback. Returns: nothing.

`audio.midi_stop()` - Stop MIDI playback. Returns: nothing.

`audio.midi_pause()` - Pause MIDI playback. Returns: nothing.

`audio.midi_resume()` - Resume MIDI playback. Returns: nothing.

`audio.midi_set_volume(0..255)` - Set global MIDI volume. Returns: nothing.

`audio.midi_is_playing()` - Check whether the MIDI player is currently playing. Returns: boolean.

`audio.midi_metadata()` - Return metadata for the currently loaded MIDI/MUS file. Returns: table with fields:

Field	Type	Description
`title`	string	Track title when present
`system`	string	`MIDI` or `Doom MUS`
`duration`	string	Duration text
`format`	string	`SMF type 0`, `SMF type 1`, or `MUS`
`tracks`	number	SMF track count
`patch_table`	string	Always `RawlandMini` in v1

Example:

```
audio.psg_load("music/track.vgm")
audio.psg_play()
sys.wait_frames(300)
local meta = audio.psg_metadata()
sys.print("Now playing:", meta.title, "by", meta.author)
audio.psg_stop()
```

`video`

Video chip, VGA, ULA, ANTIC/GTIA, TED, Voodoo 3D, Copper coprocessor, Blitter, and frame inspection.

General

`video.write_reg(addr, value)` - Write a 32-bit `value` to a video register at bus address `addr` (MMIO). Returns: nothing.

`video.read_reg(addr)` - Read a 32-bit value from a video register at bus address `addr`. Returns: number.

`video.get_dimensions()` - Get the compositor presentation dimensions. Returns: width, height (two numbers). This reports the active output frame, not necessarily an individual source's native size; for example, an enabled 800x600 Voodoo source can be aspect-fit or stretch-filled into the default 1920x1080 presentation frame.

`video.is_enabled()` - Check whether the primary VideoChip is enabled. Returns: boolean.

VGA

`video.vga_enable(on)` - Enable or disable the VGA output. Returns: nothing.

`video.vga_set_mode(mode)` - Set the VGA video mode (e.g. 0x13 for Mode 13h). Returns: nothing.

`video.vga_set_palette(idx, r, g, b)` - Set VGA palette entry `idx` (0..255) to the given RGB values (each 0..255). Returns: nothing.

`video.vga_get_palette(idx)` - Read VGA palette entry `idx`. Returns: r, g, b (three numbers, each 0..255).

`video.vga_get_dimensions()` - Get VGA framebuffer dimensions. Returns: width, height.

ULA (ZX Spectrum)

`video.ula_enable(on)` - Enable or disable ULA video output. Returns: nothing.

`video.ula_is_enabled()` - Check whether ULA is enabled. Returns: boolean.

`video.ula_border(colour)` - Set the ULA border colour (0..7). Returns: nothing.

`video.ula_get_dimensions()` - Get ULA display dimensions. Returns: width, height.

ANTIC (Atari-inspired IE-native)

`video.antic_enable(on)` - Enable or disable ANTIC video output. Returns: nothing.

`video.antic_is_enabled()` - Check whether ANTIC is enabled. Returns: boolean.

`video.antic_dlist(addr)` - Set the ANTIC display list address. Returns: nothing.

`video.antic_dma(flags)` - Set ANTIC DMA control flags (DMACTL register, 0..255). Returns: nothing.

`video.antic_scroll(h, v)` - Set ANTIC horizontal and vertical scroll values (each 0..15). Returns: nothing.

`video.antic_charset(page)` - Set ANTIC character set base page (CHBASE register, 0..255). Returns: nothing.

`video.antic_pmbase(page)` - Set ANTIC player/missile base page (PMBASE register, 0..255). Returns: nothing.

`video.antic_get_dimensions()` - Get ANTIC display dimensions. Returns: width, height.

GTIA (Atari-inspired IE-native)

`video.gtia_color(reg, value)` - Set a GTIA colour register. Register indices: 0=COLPF0, 1=COLPF1, 2=COLPF2, 3=COLPF3, 4=COLBK, 5=COLPM0, 6=COLPM1, 7=COLPM2, 8=COLPM3. Value is 0..255 (Atari hue/luminance byte). Returns: nothing.

`video.gtia_player_pos(player, x)` - Set horizontal position of player sprite `player` (0..3) to `x` (0..255). Returns: nothing.

`video.gtia_player_size(player, size)` - Set width of player sprite `player` (0..3). Size: 0=normal, 1=double, 3=quad. Returns: nothing.

`video.gtia_player_gfx(player, data)` - Set graphics data byte for player sprite `player` (0..3). Returns: nothing.

`video.gtia_priority(value)` - Set GTIA priority register (PRIOR, 0..255). Returns: nothing.

TED Video (Commodore Plus/4)

`video.ted_enable(on)` - Enable or disable TED video output. Returns: nothing.

`video.ted_is_enabled()` - Check whether TED video is enabled. Returns: boolean.

`video.ted_mode(ctrl1, ctrl2)` - Set TED control registers 1 and 2 (each 0..255). Returns: nothing.

`video.ted_colors(bg0, bg1, bg2, bg3, border)` - Set all five TED colour registers (each 0..127, TED colour format). Returns: nothing.

`video.ted_charset(page)` - Set TED character set base page (0..255). Returns: nothing.

`video.ted_video_base(page)` - Set TED video memory base page (0..255). Returns: nothing.

`video.ted_cursor(pos, colour)` - Set TED cursor position (0..65535) and colour (0..127). Returns: nothing.

`video.ted_get_dimensions()` - Get TED display dimensions. Returns: width, height.

Voodoo 3D

Voodoo functions accept integer pixel coordinates for vertices and 0..255 byte values for colours. Internally, vertex coordinates are converted to 12.4 fixed-point and colours to 12.12 fixed-point.

`video.voodoo_enable(on)` - Enable or disable Voodoo 3D rendering. Returns: nothing.

`video.voodoo_is_enabled()` - Check whether Voodoo is enabled. Returns: boolean.

`video.voodoo_resolution(w, h)` - Set the Voodoo framebuffer resolution. Returns: nothing.

`video.voodoo_vertex(ax, ay, bx, by, cx, cy)` - Set the three triangle vertex positions in integer pixel coordinates. Returns: nothing.

`video.voodoo_color(idx, r, g, b, a)` - Set vertex colour for vertex `idx` (0..2). RGBA values are 0..255. Returns: nothing.

`video.voodoo_depth(z)` - Set the starting depth value for the triangle (integer, converted to 20.12 fixed-point). Returns: nothing.

`video.voodoo_texcoord(s, t, w)` - Set texture coordinates. `s` and `t` are floating-point texture coordinates; `w` is the perspective correction factor. Returns: nothing.

`video.voodoo_draw()` - Submit the current triangle for rasterisation. Returns: nothing.

`video.voodoo_swap()` - Swap the Voodoo front and back buffers. Returns: nothing.

`video.voodoo_clear(r, g, b)` - Clear the Voodoo framebuffer with the given RGB colour (each 0..255). Returns: nothing.

`video.voodoo_fog(on, r, g, b)` - Enable/disable fog and set the fog colour. Returns: nothing.

`video.voodoo_alpha(mode)` - Set the Voodoo alpha blending mode register. Returns: nothing.

`video.voodoo_zbuffer(mode)` - Set the Voodoo depth buffer mode (FBZ_MODE register). Returns: nothing.

`video.voodoo_clip(left, right, top, bottom)` - Set the Voodoo clip rectangle. Returns: nothing.

`video.voodoo_texture(w, h, data)` - Upload texture data. `w` and `h` are the texture dimensions; `data` is a raw byte string of pixel data. Returns: nothing.

`video.voodoo_chromakey(on, r, g, b)` - Enable/disable chroma keying and set the key colour. Returns: nothing.

`video.voodoo_dither(on)` - Enable or disable dithering. Returns: nothing.

`video.voodoo_get_dimensions()` - Get Voodoo framebuffer dimensions. Returns: width, height.

Copper Coprocessor

`video.copper_enable(on)` - Enable or disable the copper coprocessor. Returns: nothing.

`video.copper_set_program(addr)` - Set the copper program pointer to bus address `addr`. Returns: nothing.

`video.copper_is_running()` - Check whether the copper is currently executing. Returns: boolean.

Blitter

`video.blit_copy(src, dst, w, h, src_stride, dst_stride)` - Start a blitter copy operation. Copies a `w`x`h` rectangle from `src` to `dst` with the given strides. Returns: nothing.

`video.blit_fill(dst, w, h, colour, dst_stride)` - Start a blitter fill operation. Fills a `w`x`h` rectangle at `dst` with `colour` (32-bit RGBA). Returns: nothing.

`video.blit_line(x0, y0, x1, y1, colour)` - Draw a line from (`x0`,`y0`) to (`x1`,`y1`) with `colour` (32-bit RGBA). Returns: nothing.

`video.blit_wait()` - Block until the blitter is idle. Polls every 1 ms. Returns: nothing.

Frame Inspection

`video.get_pixel(x, y)` - Read a pixel from the current compositor frame. Returns: r, g, b, a (four numbers, each 0..255). Returns all zeros if coordinates are out of bounds.

`video.get_region(x, y, w, h)` - Read a rectangular region from the current compositor frame. Regions that partly overlap the frame are clipped to the frame bounds. Returns: string (raw RGBA bytes, row-major, 4 bytes per pixel). Returns empty string if width or height is non-positive, no frame is available, or the requested region is entirely outside the frame.

`video.frame_hash()` - Compute an FNV-1a hash of the current compositor frame. Returns: number. Returns 0 if no frame is available.

Visual Waits

All visual waits block on the frame channel (yielding per frame) and respect script cancellation.

`video.wait_pixel(x, y, r, g, b, timeout_ms)` - Wait until the pixel at (x,y) matches the target RGB colour within a tolerance of +/-2 per channel, or until `timeout_ms` expires. Returns: boolean (`true` if matched, `false` on timeout).

`video.wait_stable(n_frames, timeout_ms)` - Wait until the compositor frame hash remains unchanged for `n_frames` consecutive frames, or until `timeout_ms` expires. Useful for waiting until rendering has settled. Returns: boolean.

`video.wait_condition(fn, timeout_ms)` - Call the Lua function `fn` once per frame. If `fn` returns `true`, the wait succeeds. Continues until `fn` returns `true` or `timeout_ms` expires. Returns: boolean.

Example - Voodoo triangle:

```
video.voodoo_enable(true)
video.voodoo_resolution(320, 240)
video.voodoo_clear(0, 0, 64)
video.voodoo_vertex(160, 10, 10, 230, 310, 230)
video.voodoo_color(0, 255, 0, 0, 255)
video.voodoo_color(1, 0, 255, 0, 255)
video.voodoo_color(2, 0, 0, 255, 255)
video.voodoo_draw()
video.voodoo_swap()
```

---

repl

Programmatic control of the Lua REPL overlay. Use this module from scripts to display information, title cards, or scrolling text on-screen without affecting the underlying emulator display.

`repl.show()` - Show the REPL overlay. Returns: nothing.

`repl.hide()` - Hide the REPL overlay. Returns: nothing.

`repl.is_open()` - Check whether the overlay is currently visible. Returns: boolean.

`repl.print(text)` - Append a line of text to the overlay output buffer. Returns: nothing.

`repl.clear()` - Clear the overlay output buffer. Returns: nothing.

`repl.scroll_up([n])` - Scroll the overlay output up by `n` lines (default 1). Returns: nothing.

`repl.scroll_down([n])` - Scroll the overlay output down by `n` lines (default 1). Returns: nothing.

`repl.line_count()` - Get the total number of lines in the overlay output buffer. Returns: number.

Example - title card:

```
repl.show()
repl.clear()
repl.print("  ================================================")
repl.print("  Intuition Engine Demo")
repl.print("  ================================================")
sys.wait_ms(3000)
repl.hide()
```

Example - scrolling source code listing:

```
local lines = {
    '10 PRINT "HELLO"',
    '20 GOTO 10',
}
repl.show(); repl.clear()
for _, line in ipairs(lines) do repl.print(line) end
repl.scroll_up(repl.line_count())
for _ = 1, repl.line_count() do
    repl.scroll_down(1)
    sys.wait_ms(60)
end
sys.wait_ms(1500)
repl.hide()
```

---

rec

Recording and screenshot capture.

`rec.screenshot(path)` - Capture the current compositor frame as a PNG file at script-relative `path`. Pure Go implementation - no external dependencies. Returns: nothing. Raises on path validation or screenshot errors.

`rec.start(path)` - Start recording video (and audio) to an MP4 file at script-relative `path`. Requires FFmpeg in `PATH`. Returns: nothing. Raises on path validation or recorder errors.

`rec.start_screen(path)` - Start recording the screen-composited output to an MP4 file at script-relative `path`. Requires FFmpeg in `PATH`. Returns: nothing. Raises on path validation or recorder errors.

`rec.stop()` - Stop an active recording and finalise the file. Returns: nothing. Raises on error.

`rec.is_recording()` - Check whether a recording is in progress. Returns: boolean.

`rec.frame_count()` - Number of frames captured in the current recording session. Returns: number.

Example:

```
rec.screenshot("before.png")
rec.start("demo.mp4")
sys.wait_frames(300)  -- record 5 seconds at 60 fps
rec.stop()
sys.print("Recorded", rec.frame_count(), "frames")
rec.screenshot("after.png")
```

---

dbg

Monitor/debugger integration. The Machine Monitor is always built into the engine - no command-line flag is required. See iemon.md for the underlying command set, breakpoint condition grammar, and CPU-specific behaviour.

Core

`dbg.open()` - Activate the monitor. The first nested `dbg.open()` / `dbg.freeze()` held by the script also increments the freeze counter. Further nested opens only increase the script debugger-open count. This is the standard way to enter a debug session from a script. Returns: nothing.

`dbg.close()` - Release one script debugger open. The monitor is deactivated and the script-owned debugger freeze is released only when the nested debugger-open count reaches zero. Extra closes beyond zero are harmless. Returns: nothing.

`dbg.is_open()` - Check whether the monitor is currently active. Returns: boolean.

`dbg.freeze()` - Alias for `dbg.open()`. Returns: nothing.

`dbg.resume()` - Alias for `dbg.close()`. Returns: nothing.

`dbg.request_break_in()` - Request an IEMon break-in on currently running CPUs. Stopped CPUs are left untouched. Returns: nothing.

Execution Control

`dbg.step([n])` - Single-step the focussed CPU by `n` instructions (default 1). Returns: nothing.

`dbg.continue()` - Execute the monitor `g` command, which deactivates IEMon and resumes the CPUs that the monitor would normally resume on exit. Returns: nothing.

`dbg.run_until(addr)` - Run the focussed CPU until it reaches address `addr`. Returns: nothing.

When `dbg.continue()` or `dbg.run_until()` resumes execution, the script-owned debugger open is released and the monitor is deactivated.

`dbg.run_until` is a fire-and-forget request: it has no timeout argument. If execution never reaches the target address before the monitor is re-entered for another reason, the temporary breakpoint set by `run_until` persists and will fire on a future run. Clear it explicitly with `dbg.clear_bp(addr)` if you no longer want the stop. See iemon.md "Common Pitfalls" for details.

`dbg.backstep()` - Restore the focussed CPU from the CPU-local step-history snapshot used by monitor `bs`. This does not rewind other CPUs or device state. Returns: nothing.

Page Guards

`dbg.guard_add(start, end, perm [, scope])` - Add a page/access guard over `start..end`. `perm` is any non-empty combination of `"r"`, `"w"`, and `"x"`. `scope` is `"all"` by default, or `"current"` for the currently focussed CPU. Returns: nothing.

`dbg.guard_del([start, end [, scope]])` - Remove a matching guard. With no arguments, clears all guards. With a range, returns the number of guards removed.

`dbg.guard_list()` - List guards. Returns a table of entries with fields `start`, `end`, `perm`, `scope`, `cpu_id`, `once`, and `name`.

Fault Interception

`dbg.fault_break(kind)` - Break before the guest handler for fault `kind` runs. For example, `dbg.fault_break("m68k.illegal")`. Returns: nothing.

`dbg.fault_clear(kind)` - Clear one fault-break kind. Returns: nothing.

`dbg.fault_on()` / `dbg.fault_off()` - Enable interception for all supported faults, or disable all fault interception. Returns: nothing.

`dbg.fault_list()` - Return `{ all = boolean, kinds = { ... } }` for the current fault-interception configuration.

`dbg.on_fault(kind, fn)` - Enable fault interception for `kind` and call `fn(event)` at script yield/poll points when that fault is observed. Use `"*"` to subscribe to all fault kinds. The event table contains `cpu_id`, `pc`, `addr`, `kind`, and `info`. Returns: nothing.

`dbg.poll_faults()` - Dispatch queued `dbg.on_fault` callbacks immediately. `sys.wait_ms` and `sys.wait_frames` also poll before returning. Returns: nothing.

Breakpoints

`dbg.set_bp(addr)` - Set an unconditional breakpoint at address `addr`. Returns: nothing.

`dbg.set_conditional_bp(addr, condition [, width])` - Set a conditional breakpoint at `addr` with condition string `condition` (e.g. `"A==$FF"`, `"[$1000]==$42"`, `"[$2000].L==$DEADBEEF"`, `"hitcount>10"`). The condition grammar is the same as the monitor `b` command; see iemon.md Breakpoints "Condition syntax" for the full operator and term reference. If `condition` starts with a memory term such as `"[$1000]"`, optional `width` may be `"B"`, `"W"`, or `"L"`; `"W"` and `"L"` insert `.W` or `.L` after the first memory term, and `"B"` leaves byte width unchanged. Returns: nothing.

`dbg.clear_bp(addr)` - Remove the breakpoint at address `addr`. Returns: nothing.

`dbg.clear_all_bp()` - Remove all breakpoints on the focussed CPU. Returns: nothing.

`dbg.list_bp()` - List all breakpoints on the focussed CPU. Returns: table (array) of entries, each with fields:

Field	Type	Description
`addr`	number	Breakpoint address
`condition`	string	Condition expression (empty if unconditional)
`hit_count`	number	Number of times this breakpoint has been hit

Watchpoints

`dbg.set_wp(addr)` - Set a one-byte write watchpoint at address `addr`. Returns: nothing.

`dbg.set_wp_ex(addr [, mode [, width]])` - Set a watchpoint with an explicit mode and width. `mode` is `"r"`, `"w"`, or `"rw"`; `width` is 1, 2, 4, or 8 bytes. Wider watchpoints fire on overlapping narrower accesses. Returns: nothing.

`dbg.clear_wp(addr)` - Remove the watchpoint at address `addr`. Returns: nothing.

`dbg.clear_all_wp()` - Remove all watchpoints on the focussed CPU. Returns: nothing.

`dbg.list_wp()` - List all watchpoint addresses. Returns: table (array) of numbers.

Symbols

`sym.add(name, addr [, kind])` - Add a symbol to the focussed CPU's symbol namespace. `kind` defaults to `"label"`. Returns: nothing.

`sym.lookup(name)` - Look up a symbol in the focussed CPU's symbol namespace. Returns: address or `nil`.

`sym.resolve(addr)` - Resolve an address to the nearest symbol. Returns a table `{name, addr, offset, kind}` or `nil`.

`sym.load_elf(path)` - Load ELF `.symtab` entries for the focussed CPU from an approved read path. Function and object symbols are imported. Returns: nothing. Raises on path validation or ELF errors.

`sym.load_vice(path [, base])` - Load VICE-style label records, or `.iesym` files that use the same accepted label syntax, from an approved read path. `base` defaults to 0 and is added to each parsed address. Returns: nothing.

`sym.load_dwarf(path)` - Load DWARF line information for the focussed CPU from an approved read path. It updates the source-line table; use `sym.load_elf(path)` as well when you also need `.symtab` symbols. Returns: nothing.

`sym.autoload(image_path [, base])` - Probe neighbouring symbol sidecars for `image_path`. `image_path` is normalised under the script roots, and every existing sidecar must pass approved read-path validation before loading. Returns `{loaded=bool, path=string|nil, kind="elf"|"guest"|nil, err=string|nil}`.

`sym.list()` - List symbols in the focussed CPU namespace. Returns a table of `{name, addr, size, kind, cpu}` entries.

Regions

`regions.list()` - List memory regions visible to the focussed CPU. Returns a table of `{start, end, name, kind}` entries.

`regions.lookup(addr)` - Resolve an address to a memory region for the focussed CPU. Returns `{start, end, name, kind}` or `nil`.

Registers

`dbg.get_reg(name)` - Read a CPU register by name (e.g. `"A"`, `"PC"`, `"SP"`). Returns: number, or `nil` if the register name is unknown.

`dbg.set_reg(name, value)` - Write a value to a CPU register by name. Returns: nothing. Raises on unknown register.

`dbg.get_regs()` - Read all CPU registers. Returns: table `{name = value, ...}`.

`dbg.get_pc()` - Read the program counter. Returns: number.

`dbg.set_pc(addr)` - Set the program counter to `addr`. Returns: nothing.

Memory

`dbg.read_mem(addr, len)` - Read `len` bytes from the focussed CPU's memory at `addr`. Returns: string (raw bytes).

`dbg.write_mem(addr, data)` - Write raw byte string `data` to the focussed CPU's memory at `addr`. Returns: nothing.

`dbg.fill_mem(addr, len, value)` - Raw 32-bit bus helper: fill `len` bytes starting at `uint32(addr)` with byte `value` through the shared bus. Addresses above `0xFFFFFFFF` are truncated to their low 32 bits before access. Returns: nothing.

`dbg.hunt_mem(start, len, pattern)` - Raw 32-bit bus helper: search for byte pattern `pattern` within `len` bytes starting at `uint32(start)` through the shared bus. Addresses above `0xFFFFFFFF` are truncated to their low 32 bits before access. Returns: table (array) of matching addresses.

`dbg.compare_mem(start, len, dest)` - Raw 32-bit bus helper: compare `len` bytes between `uint32(start)` and `uint32(dest)`, reporting differences. Addresses above `0xFFFFFFFF` are truncated to their low 32 bits before access. Returns: table (array) of entries, each with fields:

Field	Type	Description
`offset`	number	Byte offset where difference was found
`val1`	number	Byte value at `start + offset`
`val2`	number	Byte value at `dest + offset`

`dbg.transfer_mem(start, len, dest)` - Raw 32-bit bus helper: copy `len` bytes from `uint32(start)` to `uint32(dest)` through a temporary buffer, so overlapping regions are safe. Addresses above `0xFFFFFFFF` are truncated to their low 32 bits before access. Returns: nothing.

`dbg.read_mem` and `dbg.write_mem` use the focussed CPU adapter and therefore follow that CPU's address-width and adapter semantics. `dbg.fill_mem`, `dbg.hunt_mem`, `dbg.compare_mem`, and `dbg.transfer_mem` use the raw shared bus path after converting their address arguments to `uint32`; use them for bus-visible 32-bit physical ranges, not for CPU-virtual or above-4GiB IE64 addresses.

Disassembly and Trace

`dbg.disasm(addr, count)` - Disassemble `count` instructions starting at `addr`. Returns: table (array) of entries, each with fields:

Field	Type	Description
`addr`	number	Instruction address
`hex`	string	Raw instruction bytes in hex
`mnemonic`	string	Disassembled instruction text

`dbg.trace(n)` - Execute `n` instructions on the focussed CPU, recording each step. Returns: table (array) of entries, each with fields:

Field	Type	Description
`addr`	number	Instruction address
`mnemonic`	string	Disassembled instruction text
`reg_changes`	table	Register changes (currently empty table)

`dbg.backtrace([depth])` - Return a call stack backtrace up to `depth` frames (default 8). Returns: table (array) of strings.

`dbg.backtrace_frames([depth])` - Return a structured call stack backtrace up to `depth` frames (default 8, minimum 1). Raises if no monitor or CPU is available. Returns: table (array) of entries, each with fields:

Field	Type	Description
`frame`	number	Zero-based frame index
`pc`	number	Frame program counter / return address
`sym`	string/nil	Symbol name when symbol resolution succeeds; `nil` otherwise
`offset`	number	Offset from `sym`, or 0 when `sym` is `nil`

`dbg.tracering_on([size])` - Enable the focussed CPU trace ring. `size` defaults to 4096 and must be positive. Returns: nothing. Raises if no monitor or CPU is available or the monitor command reports an error.

`dbg.tracering_off()` - Disable and clear the focussed CPU trace ring. Returns: nothing. Raises if no monitor or CPU is available or the monitor command reports an error.

`dbg.tracering_show([count])` - Return the last `count` trace-ring entries for the focussed CPU. `count` defaults to 16 and must be non-negative. Returns: table (array) of entries, each with fields:

Field	Type	Description
`cpu`	string	CPU name recorded with the trace entry
`pc`	number	Program counter for the recorded instruction
`hex`	string	Raw instruction bytes in hex

Field	Type	Description
`mnemonic`	string	Disassembled instruction text

`dbg.source_at(addr)` - Resolve `addr` through the focussed CPU's loaded source map. Returns `nil` when no monitor or CPU is available or no source record covers the address. Otherwise returns a table with fields:

Field	Type	Description
`file`	string	Source file path recorded in the symbol/source map
`line`	number	One-based source line number

`dbg.trace_file(path)` - Start logging execution trace to script-relative `path`. Returns: nothing. Raises on path validation or monitor errors.

`dbg.trace_file_off()` - Stop trace file logging. Returns: nothing.

`dbg.trace_watch_add(addr)` - Add a memory address to the trace watch list. Returns: nothing.

`dbg.trace_watch_del(addr)` - Remove a memory address from the trace watch list. Returns: nothing.

`dbg.trace_watch_list()` - List all trace watch addresses. Returns: table (array) of numbers.

`dbg.trace_history(addr_str)` - Get the write history for a memory address. Pass the address as a hex string (e.g. `"$1000"`). Passing `"*"` returns an empty table (per-address query only). Returns: table (array) of entries, each with fields:

Field	Type	Description
`pc`	number	Program counter at time of write
`old_val`	number	Previous value
`new_val`	number	New value written

`dbg.trace_history_clear(addr)` - Clear write history for address `addr` (string, e.g. `"$1000"` or `"*"`). Returns: nothing.

`dbg.accesslog_on([size])` - Enable the IEMon read/write/fetch access-event ring. `size` defaults to 256 events. Raises if debug access instrumentation is unavailable. Returns: nothing.

`dbg.accesslog_off()` - Disable and clear the access-event ring. Returns: nothing.

`dbg.accesslog([count])` - Return recent access events from the ring. `count` defaults to all retained events. Each entry has fields `seq`, `cpu_id`, `pc`, `addr`, `width`, `kind`, `old_val`, and `new_val`. Returns: table.

`dbg.who(kind, addr)` - Return the most recent access event covering `addr` for `kind` (`"read"`, `"wrote"`, or `"fetched"`), or `nil`. Returns: table or nil.

`dbg.bfirst(kind, region)` - Arm a one-shot break on the first access of `kind` to the named memory region. Returns: nothing.

`dbg.reverse_continue()` - Run IEMon's whole-machine reverse-continue command (`rg`). Returns: nothing.

`dbg.reverse_until(expr)` - Run IEMon's reverse run-until command (`rt <expr>`). Returns: nothing.

`dbg.timeline([count])` - Return `tl [count]` output lines as a table of strings. Returns: table.

`dbg.history_horizon()` - Return whole-machine reverse-history retention state. Raises if no monitor or CPU is available. Returns: table with fields:

`dbg.history_horizon()` returns `snapshots`, `checkpoints`, `deltas`, `capacity`, `delta_bytes`, `checkpoint_interval`, `checkpoint_mib`, `retained_checkpoints`, and `devices`.

Field	Type	Description
`snapshots`	number	Retained snapshot count
`checkpoints`	number	Retained full checkpoint count
`deltas`	number	Retained delta count
`capacity`	number	Configured maximum retained snapshot count
`delta_bytes`	number	Approximate retained delta bytes
`checkpoint_interval`	number	Instructions between full checkpoints
`checkpoint_mib`	number	MiB cap for checkpoint storage
`retained_checkpoints`	number	Configured maximum retained checkpoint count
`devices`	number	Registered snapshot-capable device count

`dbg.history_config([opts])` - Read or update whole-machine reverse-history configuration. `opts`, when supplied, is a table with optional positive-number fields `delta_interval`, `delta_mib`, `checkpoints`, and `snapshots`. Raises if no monitor or CPU is available or any supplied value is non-positive. Returns: table with fields:

`dbg.history_config([opts])` accepts `delta_interval`, `delta_mib`, `checkpoints`, and `snapshots` as positive table fields. `dbg.history_config([opts])` returns `delta_interval`, `delta_mib`, `checkpoints`, and `snapshots`.

Field	Type	Description
`delta_interval`	number	Instructions between retained deltas
`delta_mib`	number	MiB cap for retained deltas
`checkpoints`	number	Maximum retained checkpoint count
`snapshots`	number	Maximum retained snapshot count

State Save/Load

`dbg.save_state(path)` - Save a CPU-local monitor snapshot for the currently focussed CPU to script-relative `path` by using IEMon's `ss` command. This captures that CPU adapter's registers and snapshot memory span; it does not save other CPUs, device/chip runtime state, audio/video state, timers, DMA, or monitor reverse-history state. Returns: nothing. Raises on monitor errors.

`dbg.load_state(path)` - Restore a CPU-local monitor snapshot for the currently focussed CPU from an approved read path. The snapshot CPU type must match the focussed CPU. This restores the same CPU-local scope saved by `dbg.save_state`; it does not restore whole-machine state. Use `dbg.reverse_continue()` (`rg`) or `dbg.reverse_until(expr)` (`rt <expr>`) for IEMon's whole-machine reverse-history semantics. Returns: nothing.

`dbg.save_mem_file(start, length, path)` - Save `length` bytes starting at `start` to script-relative `path`. Returns: nothing. Raises on monitor errors.

`dbg.load_mem_file(path, addr)` - Load a binary file from an approved read path into memory at `addr`. Returns: nothing. Raises on monitor errors.

Device Snapshots

`dbg.device_list()` - Return the sorted names of devices registered with IEMon's versioned snapshot service. Returns an empty table when no monitor is available.

`dbg.device_snapshot(name)` - Capture one registered device snapshot. Raises if no monitor is available or the device snapshot callback fails. Returns `nil` when `name` is not registered. Otherwise returns a table with fields:

Field	Type	Description
`name`	string	Device name passed to `dbg.device_snapshot`
`version`	number	Device snapshot format version
`data`	string	Opaque byte string containing the device snapshot payload

`dbg.device_diff(a, b)` - Compare two device snapshot tables, usually returned by `dbg.device_snapshot`. Arguments must contain `name`, `version`, and `data`. Raises an argument error if either table is missing those fields. Returns: string diff summary.

Multi-CPU

`dbg.cpu_list()` - List all registered CPUs. Returns: table (array) of entries, each with fields:

Field	Type	Description
`id`	number	CPU identifier
`label`	string	CPU label/name
`cpu_name`	string	CPU architecture name
`is_running`	boolean	Whether the CPU is currently running

`dbg.cpu_focus(id)` - Switch monitor focus to a CPU by numeric `id` or string label. Returns: nothing.

`dbg.cpu_online(type_or_path [, path_or_replace] [, replace])` - Start an offline coprocessor worker through monitor `cpu online`. The first argument is either a CPU type (`"ie32"`, `"6502"`, `"m68k"`, `"z80"`, `"x86"`, or `"ie64"`) or a typed worker image path accepted by IEMon. If the second argument is a string, it is validated as an approved read path and passed as the worker image; if it is a boolean, it is treated as `replace`. The third argument, when present, is the boolean `replace` flag. Returns: nothing. Raises on monitor, path validation, or worker start errors.

`dbg.cpu_offline(id_or_label)` - Stop an online coprocessor worker by numeric ID, label, or type through monitor `cpu offline`. Returns: nothing. Raises on monitor errors.

`dbg.freeze_cpu(label)` - Freeze a specific CPU by label. Returns: nothing.

`dbg.thaw_cpu(label)` - Thaw (resume) a specific CPU by label. Returns: nothing.

`dbg.freeze_all()` - Freeze all CPUs. Returns: nothing.

`dbg.thaw_all()` - Thaw all CPUs. Returns: nothing.

Audio Debug

`dbg.freeze_audio()` - Freeze audio generation (silence). Returns: nothing.

`dbg.thaw_audio()` - Resume audio generation. Returns: nothing.

I/O Inspection

`dbg.io_devices()` - List all available I/O device names. Returns: table (array) of strings.

`dbg.io(device)` - Read all registers for the named I/O device. If `device` is not a recognised device name (see `dbg.io_devices()` for the authoritative list), returns an empty table silently - typos do not raise. Returns: table (array) of

entries, each with fields:

Field	Type	Description
`name`	string	Register name
`addr`	number	Register address
`value`	number	Current register value
`access`	string	Access mode (e.g. `"RW"`, `"RO"`)

`dbg.mmio_stats()` - `dbg.mmio_stats()` returns rows with `start`, `end`, `name`, `reads`, and `writes`. Counters are recorded only when `IE_MMIO_STATS=1` was set at process start. When the bus is unavailable, returns an empty table. Returns: table (array) of entries, each with fields:

Field	Type	Description
`start`	number	Inclusive mapped-region start address
`end`	number	Inclusive mapped-region end address
`name`	string	I/O device name when the region can be matched, otherwise empty
`reads`	number	Recorded 32-bit read count
`writes`	number	Recorded 32-bit write count

Scripting

`dbg.run_script(path)` - Execute a monitor script file from an approved read path. Before execution, every line and semicolon-separated command is validated with the same sandbox filter used by `dbg.command`. Host-file monitor commands are rejected. Returns: nothing.

`dbg.macro(name, cmds)` - Define a monitor macro. `name` is the macro name; `cmds` is the semicolon-aware command string. Each command is sandbox-validated before registration. Macro names cannot contain whitespace or semicolons. Returns: nothing.

`dbg.layout(name)` - Run IEMon `layout <name>` and return the monitor output as one string joined with newlines. Raises if no monitor is available or the monitor command reports an error.

`dbg.bug_report([trace_count])` - Run IEMon `bug <trace_count>` and return the report text as one newline-joined string. `trace_count` defaults to 16 and must be positive. Raises if no monitor is available or the monitor command reports an error.

`dbg.help([topic])` - Run IEMon `help` or `help <topic>` and return the help text as one newline-joined string. Raises if no monitor is available or the monitor command reports an error.

`dbg.command(cmd)` - Execute a monitor command string after sandbox filtering. Host-file-capable monitor commands are rejected (`save`, `load`, `ss`, `sl`, `script`, `macro`, and `trace file`; `trace file off` is allowed). Invoking monitor macros through this raw API is rejected. Returns: nothing.

`dbg.command_output(cmd)` - Execute a sandbox-filtered monitor command string and return newly appended monitor output lines as `{text, color}` entries. The field is named `color` because that is the exported Lua table key. The same command restrictions as `dbg.command` apply. Returns: table.

The raw monitor wrappers cannot enter or feed IEMon IE64 assemble mode. `A <addr>` is interactive-only, so `dbg.command`, `dbg.command_output`, `dbg.run_script`, and `dbg.macro` reject it and also reject attempts to send instruction text while the monitor is already in assemble mode.

Example - breakpoint workflow:

```
dbg.open()
dbg.set_bp(0x1000)
dbg.continue()
-- execution stops at breakpoint
local pc = dbg.get_pc()
sys.print("Stopped at:", string.format("$%04X", pc))
local regs = dbg.get_regs()
for name, val in pairs(regs) do
  sys.print(name, "=", string.format("$%X", val))
end
local dis = dbg.disasm(pc, 5)
for _, entry in ipairs(dis) do
  sys.print(string.format("$%04X  %s  %s", entry.addr, entry.hex, entry.mnemonic))
end
dbg.clear_all_bp()
dbg.close()
```

---

coproc

Coprocessor manager for offloading work to secondary CPU instances. This section documents the Lua-facing ticket API; script callers do not need the low-level MMIO and mailbox contract.

Supported CPU types: `"ie32"`, `"6502"`, `"m68k"`, `"z80"`, `"x86"`, `"ie64"`.

Ticket lifecycle

1. `coproc.start(cpu_type, filename)` - launch a worker.
2. `coproc.enqueue(cpu_type, op, request)` - submit work, get a ticket ID.
3. `coproc.poll(ticket)` or `coproc.wait(ticket, timeout_ms)` - check/wait for completion.
4. `coproc.response(ticket)` - retrieve the response data.
5. `coproc.stop(cpu_type)` - tear down the worker when done.

Functions

`coproc.start(cpu_type, filename)` - Start a coprocessor worker of the given `cpu_type`, loading the program from `filename`. `filename` is resolved by the coprocessor manager relative to its configured base directory; absolute paths and names containing `..` are rejected. Returns: nothing. Raises on error.

`coproc.stop(cpu_type)` - Stop the coprocessor worker for `cpu_type`. Returns: nothing. Raises on error.

`coproc.enqueue(cpu_type, op, request)` - Enqueue a work request. `op` is a numeric opcode; `request` is a raw byte string payload. Returns: number (ticket ID).

`coproc.poll(ticket)` - Check the status of a ticket without blocking. Returns: string - one of `"pending"`, `"running"`, `"ok"`, `"error"`, `"timeout"`, `"worker_down"`.

`coproc.wait(ticket, timeout_ms)` - Block until the ticket completes or `timeout_ms` expires. Returns: status (string), response (string, raw bytes). The response is empty if the ticket did not complete successfully.

`coproc.workers()` - List all active coprocessor workers. Returns: table (array) of entries, each with fields:

Field	Type	Description
`cpu_type`	string	CPU type name
`is_running`	boolean	Whether the worker is active

Per-CPU monitor registers such as ring depth and uptime are selected by writing `COPROC_CPU_TYPE` before reading the register. `COPROC_BUSY_PCT` is aggregate across workers. For 6502 and Z80 workers, the mailbox CPU window is `0x2000` through `0x37FF`; `0x3800` through `0x3FFF` remains worker RAM.

`coproc.response(ticket)` - Retrieve the response data for a ticket. If the ticket completed successfully, returns the response bytes. If the ticket is not found in the response ring but was previously enqueued, returns the raw contents of the preallocated response buffer (which may contain stale or partial data). Returns empty string only if the ticket is entirely unknown. Returns: string (raw bytes).

`coproc.stats()` - Read aggregate dispatcher counters from MMIO. Returns: table with fields:

Field	Type	Source register
`ops`	number	`COPROC_STATS_OPS` - total completed ops
`bytes`	number	`COPROC_STATS_BYTES` - total bytes processed
`overhead_ns`	number	`COPROC_DISPATCH_OVERHEAD` - most recent dispatch overhead in ns
`completed_ticket`	number	`COPROC_COMPLETED_TICKET` - most recently completed ticket ID

Example:

```
coproc.start("ie32", "worker.ie32")
local ticket = coproc.enqueue("ie32", 1, "input data")
local status, response = coproc.wait(ticket, 5000)
sys.print("Status:", status, "Response length:", #response)
coproc.stop("ie32")
```

`media`

Format-agnostic media loader. Supports SID, PSG/VGM, TED, AHX, POKEY/SAP, MOD, and WAV formats. WAV routing supports mono/stereo PCM through the WAV MMIO control surface. MIDI routing supports SMF `.mid`/`.midi` and Doom `.mus` through the fixed built-in RawlandMini IE SoundChip GM-style/chiptune interpretation. It is not GM hardware emulation, with up to 10 active MIDI voices and deterministic voice stealing.

`media.load(filename)` - Load and start playing a music file from an approved read path, auto-detecting format. Returns: nothing. Raises on path validation or immediate setup failures (e.g. scratch memory unavailable); format detection and decode errors are reported asynchronously via `media.status()` and `media.error()`.

`media.load_subsong(filename, subsong)` - Load a music file from an approved read path and select a specific sub-song index. Returns: nothing. Same error semantics as `media.load`.

`media.play()` - Resume playback (if paused or after load). Returns: nothing.

`media.stop()` - Stop playback. Returns: nothing.

`media.status()` - Get the current playback status. Returns: string - one of `"idle"`, `"loading"`, `"playing"`, `"error"`.

`media.type()` - Get the detected media type. `media.type() returns sid, psg, ted, ahx, pokey, mod, wav, midi, or none`. Returns: string.

`media.error()` - Get the last error code (0 if no error). Returns: number.

Example:

```
media.load("music/song.sid")
sys.wait_frames(60)
sys.print("Playing:", media.type(), "Status:", media.status())
sys.wait_frames(600)
media.stop()
```

---

bit32

Lua 5.1 does not include a bitwise library. IEScript provides a `bit32` global table with unsigned 32-bit operations, compatible with the Lua 5.2 `bit32` library interface.

`bit32.band(...)` - Bitwise AND of all arguments. With zero arguments, returns `0xFFFFFFFF`. Returns: number.

`bit32.bor(...)` - Bitwise OR of all arguments. Returns: number.

`bit32.bxor(...)` - Bitwise XOR of all arguments. Returns: number.

`bit32.bnot(x)` - Bitwise NOT (ones complement). Returns: number.

`bit32.lshift(x, disp)` - Logical left shift by `disp` bits. `bit32.lshift(x, disp) masks disp to 0..31 and returns number`.

`bit32.rshift(x, disp)` - Logical right shift by `disp` bits. `bit32.rshift(x, disp) masks disp to 0..31 and returns number`.

`bit32.arshift(x, disp)` - Arithmetic right shift by `disp` bits. `bit32.arshift(x, disp) masks disp to 0..31, sign-extends, and returns number`.

`bit32.lrotate(x, disp)` - Left rotation by `disp` bits. `bit32.lrotate(x, disp) masks disp to 0..31 and returns number`.

`bit32.rrotate(x, disp)` - Right rotation by `disp` bits. `bit32.rrotate(x, disp) masks disp to 0..31 and returns number`.

`bit32.extract(x, field[, width])` - Extract `width` bits starting at zero-based bit `field` (default width 1). `bit32.extract(x, field[, width]) raises an error for field < 0, width <= 0, or field + width > 32`. Returns: number.

`bit32.replace(x, v, field[, width])` - Replace `width` bits in `x` starting at zero-based bit `field` with low bits from `v` (default width 1). `bit32.replace(x, v, field[, width]) raises an error for field < 0, width <= 0, or field + width > 32`. Returns: number.

`bit32.btest(...)` - Bitwise AND of all arguments. `bit32.btest(...) returns boolean true when the bitwise AND result is non-zero`.

Example:

```
local flags = bit32.bor(0x01, 0x04, 0x10)   -- 0x15
local masked = bit32.band(flags, 0x0F)       -- 0x05
local shifted = bit32.lshift(1, 7)           -- 0x80
sys.print(string.format("0x%X", shifted))
```

Recording and Screenshots

Screenshot

```
rec.screenshot("frame.png")
```

Screenshots are pure Go (PNG encoding) - no external tools required.

Recording

```
rec.start("demo.mp4")
sys.wait_frames(300)
rec.stop()
```

Notes:

- FFmpeg must be available in `PATH`.
- Recording uses compositor dimensions/refresh settings.
- Audio is captured via a sample tap on the sound chip - no double-ticking occurs.
- Resolution is locked for the duration of a recording session.
- Recording works in headless builds; use the normal `-script render.ies` option on a binary built with the `headless` tag.

Full demo recording workflow

```
-- Load a program and record a 10-second demo
cpu.load("demo.ie32")
cpu.start()
sys.wait_frames(30)  -- let the demo initialise

rec.start("output.mp4")
sys.wait_frames(600)  -- 10 seconds at 60 fps
rec.stop()

rec.screenshot("final_frame.png")
sys.quit()
```

Lua REPL Overlay (F8)

Press `F8` to open/close the Lua REPL overlay. The REPL shares the same Lua API as scripts.

Keyboard shortcuts

Key	Action
`F8`	Toggle overlay open/close
`Esc`	Close overlay
`Enter`	Execute current line
`Up` / `Down`	Command history navigation
`Ctrl+A`	Move cursor to start of line
`Ctrl+E`	Move cursor to end of line
`Ctrl+K`	Kill text from cursor to end of line
`Ctrl+U`	Kill text from start of line to cursor
`PgUp` / `PgDn`	Scroll output buffer
`Ctrl+Shift+V`	Paste from clipboard

Expression shortcut

Type `=expr` as a shortcut for `return expr`:

```
> =sys.fps()
60
> =cpu.mode()
ie64
```

Multiline input

Incomplete chunks (e.g. an unclosed `function ... end` block) trigger a continuation prompt, allowing multiline input.

Headless builds

The REPL overlay is not available in headless builds (no display backend). Use `-script` for headless automation instead.

EhBASIC Integration

`RUN "file.ies"` routes through ProgramExecutor `.ies` detection and runs the file through the same script engine used by `-script`. This allows script execution without firmware keyword changes.

Worked Examples

Basic automation and monitor commands

```
term.type_line('PRINT "HELLO FROM LUA"')
term.wait_output("HELLO FROM LUA", 2000)

dbg.open()
dbg.command("r")
dbg.close()
```

Visual wait

```
local ok = video.wait_pixel(10, 10, 255, 0, 0, 3000)
if not ok then
  error("pixel did not reach target colour in time")
end
```

Voodoo quick draw

```
video.voodoo_enable(true)
video.voodoo_resolution(320, 240)
video.voodoo_clear(0, 0, 64)
video.voodoo_vertex(160, 10, 10, 230, 310, 230)
video.voodoo_color(0, 255, 0, 0, 255)
video.voodoo_color(1, 0, 255, 0, 255)
video.voodoo_color(2, 0, 0, 255, 255)
video.voodoo_draw()
video.voodoo_swap()
```

Full demo recording

```
cpu.load("demo.ie32")
cpu.start()
sys.wait_frames(30)

rec.start("demo.mp4")
sys.wait_frames(600)
rec.stop()

rec.screenshot("final.png")
sys.quit()
```

Monitor debugging workflow

```
dbg.open()

-- Set a breakpoint and run to it
dbg.set_bp(0x1000)
dbg.continue()

-- Inspect state at breakpoint
local pc = dbg.get_pc()
sys.print("Hit breakpoint at:", string.format("$%04X", pc))

-- Disassemble around the breakpoint
local dis = dbg.disasm(pc, 10)
for _, d in ipairs(dis) do
  sys.print(string.format("  $%04X  %-12s  %s", d.addr, d.hex, d.mnemonic))
end

-- Read registers
local regs = dbg.get_regs()
for name, val in pairs(regs) do
  sys.print(string.format("  %s = $%X", name, val))
end

-- Single-step a few instructions
dbg.step(3)
sys.print("After 3 steps, PC =", string.format("$%04X", dbg.get_pc()))

-- Clean up
dbg.clear_all_bp()
dbg.close()
```

Troubleshooting

`raw memory access requires cpu.freeze()`

Wrap RAM operations with `cpu.freeze()` and `cpu.resume()`.

`ffmpeg not found in PATH`

Install FFmpeg and ensure the executable is resolvable from your shell session.

Script appears stalled

- check waits and timeouts (`wait_frames`, `wait_ms`, visual waits)
- print state periodically with `sys.print`
- inspect monitor state via `dbg.command(...)` for sandbox-safe monitor commands

REPL prints but script output not visible

Use `sys.print` for host console output and keep REPL open for in-overlay logs.

REPL overlay not appearing

The overlay requires a display backend. It is not available in headless builds (`-tags headless` or `make headless`). Use `-script` for headless automation.

Recording stops unexpectedly

Recording relies on an FFmpeg subprocess. If FFmpeg crashes or is killed, the recording stops. Check FFmpeg stderr output for encoding errors. Common causes: unsupported resolution, disk full, or codec issues.

Script Cancellation and Auto-Release

When a script raises an unhandled error or is cancelled (by host shutdown, Lua VM context cancellation, or an explicit stop), the runtime auto-releases the script's contributions to global state so the emulator returns to a coherent baseline. The deferred cleanup in `ScriptEngine.run` performs:

- **CPU freeze counter** - every outstanding `cpu.freeze()` made by the script is decremented.
- **Debugger open count** - if the script holds the script-owned `dbg.open()` / `dbg.freeze()` contribution, the monitor is deactivated and the associated CPU freeze it added is released.
- **Audio freeze** - if the script called `audio.freeze()` or `audio.resume()`, the sound chip's `audioFrozen` flag is restored to the value it had at script start.
- **Coprocessor tickets** - the in-process ticket table is cleared.
- **Output capture** - `sys.capture_output` redirection is reverted.
- **Mouse override** - the sticky flag set by `term.mouse_*` injection is cleared so host mouse handling resumes.

State *not* auto-released: breakpoints, watchpoints, monitor macros, trace watches, `dbg.run_until` temp breakpoints, audio changes made through `dbg.freeze_audio()` / `dbg.thaw_audio()`, any guest-side state mutated through `mem.*` / `dbg.write_mem`, and **active recordings started by `rec.start*`** (recording stops only on explicit `rec.stop()`, `sys.quit()`, `sys.exit()`, replacement by a new script, explicit cancellation, or engine shutdown). Clean these up explicitly when correctness depends on it.

Common Pitfalls

- **`raw memory access requires cpu.freeze()`** - every `mem.read*` / `mem.write*` / `mem.read_block` / `mem.write_block` / `mem.fill` on a RAM address must be inside a `cpu.freeze()` / `cpu.resume()` bracket. MMIO addresses are exempt, but block operations that span MMIO into RAM still require a freeze.
- **`audio.*_load` format mismatch** - each player accepts only its own file types. Use the format-agnostic `media.load` if you need auto-detection across SID, PSG/VGM, TED, AHX, POKEY, MOD, WAV, and MIDI/MUS.
- **Host-FS denial outside script roots** - relative reads search the current script directory then `sdk/scripts/`; absolute reads succeed only when the resolved target remains under an approved root. Writes are script-relative only. Traversal and symlink escapes are rejected.
- **`require` only loads Lua modules from approved roots** - native modules and `package.loadlib` are unavailable.
- **`rec.start*` needs FFmpeg in `PATH`** - `rec.screenshot` is pure Go and has no external dependency.
- **`dbg.run_until` has no timeout** - leaves a temp breakpoint if the target is never reached. Clear with `dbg.clear_bp(addr)`.
- **`dbg.io(device)` returns an empty table for unknown device names** - no error is raised; check `dbg.io_devices()` for the canonical list.
- **`cpu.set_jit_enabled(true)` raises while the CPU is running** - stop the CPU first or toggle JIT only at boot.

- **Frame channel capacity 1 -** if inter-yield work exceeds a frame period, frames are silently dropped rather than queued. Inspect `sys.frame_time()` to detect.
- **`term.mouse_*` injection sets a sticky override -** call `term.mouse_release()` when done so host mouse handling resumes.

Quick Reference

Compact reference for IEScript API functions.

sys (19)

Function	Returns
`sys.wait_frames(n)`	-
`sys.wait_ms(ms)`	-
`sys.print(...)`	-
`sys.log(...)`	-
`sys.time_ms()`	number
`sys.frame_count()`	number
`sys.frame_time()`	number
`sys.fps()`	number
`sys.perf_report()`	string
`sys.perf_reset()`	-
`sys.quit()`	-
`sys.exit([code])`	-
`sys.emutos_drive(path [, drive])`	-
`sys.mkdir(path)`	string
`sys.read_file(path)`	string
`sys.write_file(path, data)`	-
`sys.copy_file(src, dst)`	string
`sys.capture_output(path)`	-
`sys.capture_output_off()`	-

cpu (13)

Function	Returns
`cpu.load(path)`	-
`cpu.load_stopped(path)`	-
`cpu.reset()`	-
`cpu.freeze()`	-

Function	Returns
`cpu.resume()`	-
`cpu.start()`	-
`cpu.stop()`	-
`cpu.is_running()`	boolean
`cpu.mode()`	string
`cpu.jit_enabled()`	boolean
`cpu.set_jit_enabled(enabled)`	-
`cpu.execution_mode()`	string
`cpu.jit_stats()`	table

mem (9)

Function	Returns
`mem.read8(addr)`	number
`mem.read16(addr)`	number
`mem.read32(addr)`	number
`mem.write8(addr, value)`	-
`mem.write16(addr, value)`	-
`mem.write32(addr, value)`	-
`mem.read_block(addr, len)`	string
`mem.write_block(addr, bytes)`	-
`mem.fill(addr, len, value)`	-

term (14)

Function	Returns
`term.type(str)`	-
`term.type_line(str)`	-
`term.read()`	string
`term.clear()`	-
`term.echo(on)`	-
`term.wait_output(pattern, timeout_ms)`	boolean
`term.mouse_move(x, y)`	-
`term.mouse_delta(dx, dy [, button])`	-
`term.mouse_click(x, y [, button])`	-
`term.mouse_press(x, y [, button])`	-

Function	Returns
`term.mouse_double_click(x, y [, button])`	-
`term.mouse_release()`	-
`term.scancode(code)`	-
`term.key_press(code [, hold_ms])`	-

keys

`keys` is a global table of Atari ST make-code constants for use with `term.scancode()` and `term.key_press()`. It contains `ESCAPE`, `BACKSPACE`, `TAB`, `ENTER`, `SPACE`, `LSHIFT`, `RSHIFT`, `LCTRL`, `CAPSLOCK`, `F1` through `F10`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `A` through `Z`, `DIGIT_0` through `DIGIT_9`, `MINUS`, and `EQUAL`.

audio (44)

Function	Returns
`audio.start()`	-
`audio.stop()`	-
`audio.reset()`	-
`audio.freeze()`	-
`audio.resume()`	-
`audio.write_reg(addr, value)`	-
`audio.set_master_gain_db(db)`	-
`audio.get_master_gain_db()`	number
`audio.set_master_auto_level_enabled(on)`	-
`audio.configure_master_auto_level(target_db, min_gain_db, max_gain_db, attack_ms, release_ms)`	-
`audio.set_master_compressor_enabled(on)`	-
`audio.configure_master_compressor(threshold_db, ratio, attack_ms, release_ms, knee_db, makeup_db, lookahead_ms)`	-
`audio.use_showreel_normalizer_preset()`	-
`audio.reset_master_dynamics()`	-
`audio.psg_load(path)`	-
`audio.psg_play()`	-
`audio.psg_stop()`	-
`audio.psg_is_playing()`	boolean
`audio.psg_metadata()`	table
`audio.sid_load(path [, subsong])`	-
`audio.sid_play()`	-

Function	Returns
`audio.sid_stop()`	-
`audio.sid_is_playing()`	boolean
`audio.sid_metadata()`	table
`audio.ted_load(path)`	-
`audio.ted_play()`	-
`audio.ted_stop()`	-
`audio.ted_is_playing()`	boolean
`audio.pokey_load(path)`	-
`audio.pokey_play()`	-
`audio.pokey_stop()`	-
`audio.pokey_is_playing()`	boolean
`audio.ahx_load(path)`	-
`audio.ahx_play()`	-
`audio.ahx_stop()`	-
`audio.ahx_is_playing()`	boolean
`audio.midi_load(path)`	-
`audio.midi_play()`	-
`audio.midi_stop()`	-
`audio.midi_pause()`	-
`audio.midi_resume()`	-
`audio.midi_set_volume(0..255)`	-
`audio.midi_is_playing()`	boolean
`audio.midi_metadata()`	table

video (65)

Function	Returns
`video.write_reg(addr, value)`	-
`video.read_reg(addr)`	number
`video.get_dimensions()`	width, height
`video.is_enabled()`	boolean
`video.vga_enable(on)`	-
`video.vga_set_mode(mode)`	-
`video.vga_set_palette(idx, r, g, b)`	-
`video.vga_get_palette(idx)`	r, g, b

Function	Returns
`video.vga_get_dimensions()`	width, height
`video.ula_enable(on)`	-
`video.ula_is_enabled()`	boolean
`video.ula_border(colour)`	-
`video.ula_get_dimensions()`	width, height
`video.antic_enable(on)`	-
`video.antic_is_enabled()`	boolean
`video.antic_dlist(addr)`	-
`video.antic_dma(flags)`	-
`video.antic_scroll(h, v)`	-
`video.antic_charset(page)`	-
`video.antic_pmbase(page)`	-
`video.antic_get_dimensions()`	width, height
`video.gtia_color(reg, value)`	-
`video.gtia_player_pos(player, x)`	-
`video.gtia_player_size(player, size)`	-
`video.gtia_player_gfx(player, data)`	-
`video.gtia_priority(value)`	-
`video.ted_enable(on)`	-
`video.ted_is_enabled()`	boolean
`video.ted_mode(ctrl1, ctrl2)`	-
`video.ted_colors(bg0, bg1, bg2, bg3, border)`	-
`video.ted_charset(page)`	-
`video.ted_video_base(page)`	-
`video.ted_cursor(pos, colour)`	-
`video.ted_get_dimensions()`	width, height
`video.voodoo_enable(on)`	-
`video.voodoo_is_enabled()`	boolean
`video.voodoo_resolution(w, h)`	-
`video.voodoo_vertex(ax, ay, bx, by, cx, cy)`	-
`video.voodoo_color(idx, r, g, b, a)`	-
`video.voodoo_depth(z)`	-
`video.voodoo_texcoord(s, t, w)`	-
`video.voodoo_draw()`	-

Function	Returns
`video.voodoo_swap()`	-
`video.voodoo_clear(r, g, b)`	-
`video.voodoo_fog(on, r, g, b)`	-
`video.voodoo_alpha(mode)`	-
`video.voodoo_zbuffer(mode)`	-
`video.voodoo_clip(left, right, top, bottom)`	-
`video.voodoo_texture(w, h, data)`	-
`video.voodoo_chromakey(on, r, g, b)`	-
`video.voodoo_dither(on)`	-
`video.voodoo_get_dimensions()`	width, height
`video.copper_enable(on)`	-
`video.copper_set_program(addr)`	-
`video.copper_is_running()`	boolean
`video.blit_copy(src, dst, w, h, src_stride, dst_stride)`	-
`video.blit_fill(dst, w, h, colour, dst_stride)`	-
`video.blit_line(x0, y0, x1, y1, colour)`	-
`video.blit_wait()`	-
`video.get_pixel(x, y)`	r, g, b, a
`video.get_region(x, y, w, h)`	string
`video.frame_hash()`	number
`video.wait_pixel(x, y, r, g, b, timeout_ms)`	boolean
`video.wait_stable(n_frames, timeout_ms)`	boolean
`video.wait_condition(fn, timeout_ms)`	boolean

repl (8)

Function	Returns
`repl.show()`	-
`repl.hide()`	-
`repl.is_open()`	boolean
`repl.print(text)`	-
`repl.clear()`	-
`repl.scroll_up([n])`	-
`repl.scroll_down([n])`	-
`repl.line_count()`	number

rec (6)

Function	Returns
`rec.screenshot(path)`	-
`rec.start(path)`	-
`rec.start_screen(path)`	-
`rec.stop()`	-
`rec.is_recording()`	boolean
`rec.frame_count()`	number

dbg

Function	Returns
`dbg.open()`	-
`dbg.close()`	-
`dbg.is_open()`	boolean
`dbg.freeze()`	-
`dbg.resume()`	-
`dbg.request_break_in()`	-
`dbg.step([n])`	-
`dbg.continue()`	-
`dbg.run_until(addr)`	-
`dbg.backstep()`	-
`dbg.set_bp(addr)`	-
`dbg.set_conditional_bp(addr, condition [, width])`	-
`dbg.clear_bp(addr)`	-
`dbg.clear_all_bp()`	-
`dbg.list_bp()`	table
`dbg.set_wp(addr)`	-
`dbg.set_wp_ex(addr [, mode [, width]])`	-
`dbg.clear_wp(addr)`	-
`dbg.clear_all_wp()`	-
`dbg.list_wp()`	table
`dbg.get_reg(name)`	number/nil
`dbg.set_reg(name, value)`	-
`dbg.get_regs()`	table
`dbg.get_pc()`	number

Function	Returns
dbg.set_pc(addr)	-
dbg.read_mem(addr, len)	string
dbg.write_mem(addr, data)	-
dbg.fill_mem(addr, len, value)	-
dbg.hunt_mem(start, len, pattern)	table
dbg.compare_mem(start, len, dest)	table
dbg.transfer_mem(start, len, dest)	-
dbg.backtrace([depth])	table
dbg.backtrace_frames([depth])	table
dbg.disasm(addr, count)	table
dbg.trace(n)	table
dbg.tracering_on([size])	-
dbg.tracering_off()	-
dbg.tracering_show([count])	table
dbg.source_at(addr)	table/nil
dbg.trace_file(path)	-
dbg.trace_file_off()	-
dbg.trace_watch_add(addr)	-
dbg.trace_watch_del(addr)	-
dbg.trace_watch_list()	table
dbg.trace_history(addr_str)	table
dbg.trace_history_clear(addr)	-
dbg.accesslog_on([size])	-
dbg.accesslog_off()	-
dbg.accesslog([count])	table
dbg.who(kind, addr)	table/nil
dbg.bfirst(kind, region)	-
dbg.reverse_continue()	-
dbg.reverse_until(expr)	-
dbg.timeline([count])	table
dbg.history_horizon()	table
dbg.history_config([opts])	table
dbg.guard_add(start, end, perm [, scope])	-
dbg.guard_del([start, end [, scope]])	number

Function	Returns
`dbg.guard_list()`	table
`dbg.fault_on()` / `dbg.fault_off()`	-
`dbg.fault_break(kind)` / `dbg.fault_clear(kind)`	-
`dbg.fault_list()`	table
`dbg.on_fault(kind, fn)`	-
`dbg.poll_faults()`	-
`dbg.save_state(path)`	-
`dbg.load_state(path)`	-
`dbg.save_mem_file(start, length, path)`	-
`dbg.load_mem_file(path, addr)`	-
`dbg.device_list()`	table
`dbg.device_snapshot(name)`	table/nil
`dbg.device_diff(a, b)`	string
`dbg.cpu_list()`	table
`dbg.cpu_focus(id)`	-
`dbg.cpu_online(type_or_path [, path_or_replace] [, replace])`	-
`dbg.cpu_offline(id_or_label)`	-
`dbg.freeze_cpu(label)`	-
`dbg.thaw_cpu(label)`	-
`dbg.freeze_all()`	-
`dbg.thaw_all()`	-
`dbg.freeze_audio()`	-
`dbg.thaw_audio()`	-
`dbg.io_devices()`	table
`dbg.io(device)`	table
`dbg.mmio_stats()`	table
`dbg.run_script(path)`	-
`dbg.macro(name, cmds)`	-
`dbg.layout(name)`	string
`dbg.bug_report([trace_count])`	string
`dbg.help([name])`	string
`dbg.command(cmd)`	-
`dbg.command_output(cmd)`	table

New monitor-parity wrappers return structured data where the monitor prints text. `dbg.tracering_show()` returns `{cpu, pc, hex, mnemonic}` entries. `dbg.backtrace_frames()` returns `{frame, pc, sym, offset}` entries and leaves `dbg.backtrace()` text-compatible. `dbg.device_snapshot()` returns `{name, version, data}` with `data` as an opaque byte string; compare two snapshots with `dbg.device_diff()`. `dbg.history_config({delta_interval=32, delta_mib=64, checkpoints=8, snapshots=256})` pins reverse-history retention for deterministic scripts.

`dbg.on_fault(kind, fn)` receives `{kind, cpu_id, pc, addr, info}`. `info` is CPU-specific flat text: IE64 privilege/illegal/syscall faults include the decoded trap or control-register context when available; M68K/Z80/6502/x86 faults report the adapter-supplied fault detail; guard and access faults include the monitored address/range text emitted by the debug service.

sym

Function	Returns
`sym.add(name, addr [, kind])`	-
`sym.lookup(name)`	number/nil
`sym.resolve(addr)`	table/nil
`sym.load_elf(path)`	-
`sym.load_vice(path [, base])`	-
`sym.autoload(image_path [, base])`	table
`sym.load_dwarf(path)`	-
`sym.list()`	table

`sym.autoload()` probes `<image>.elf`, `<stem>.elf`, then guest label sidecars `<image>.iesym`, `<image>.lbl`, `<stem>.iesym`, `<stem>.lbl`. It returns `{loaded=bool, path=string|nil, kind="elf"|"guest"|nil, err=string|nil}`. The returned `path`, when non-nil, is the validated sidecar path that was actually loaded or rejected.

regions (2)

Function	Returns
`regions.list()`	table
`regions.lookup(addr)`	table/nil

coproc (8)

Function	Returns
`coproc.start(cpu_type, filename)`	-
`coproc.stop(cpu_type)`	-
`coproc.enqueue(cpu_type, op, request)`	number (ticket)
`coproc.poll(ticket)`	string
`coproc.wait(ticket, timeout_ms)`	string, string
`coproc.workers()`	table
`coproc.response(ticket)`	string

Function	Returns
`coproc.stats()`	table

media (7)

Function	Returns
`media.load(filename)`	-
`media.load_subsong(filename, subsong)`	-
`media.play()`	-
`media.stop()`	-
`media.status()`	string
`media.type()`	string
`media.error()`	number

bit32 (12)

Function	Returns
`bit32.band(...)`	number
`bit32.bor(...)`	number
`bit32.bxor(...)`	number
`bit32.bnot(x)`	number
`bit32.lshift(x, disp)`	number
`bit32.rshift(x, disp)`	number
`bit32.arshift(x, disp)`	number
`bit32.lrotate(x, disp)`	number
`bit32.rrotate(x, disp)`	number
`bit32.extract(x, field[, width])`	number
`bit32.replace(x, v, field[, width])`	number
`bit32.btest(...)`	boolean