Intuition Engine Machine Monitor

Last modified: 2026-07-07

Overview

The Machine Monitor is a built-in hardware-level debugger inspired by the Commodore 64/Amiga Action Replay cartridge, HRTMon, the Commodore Plus/4 built-in monitor, NuMega SoftICE, MAME's debugger, and the VICE monitor. In interactive display builds, press **F9** to freeze the guest CPUs and enter the monitor. Press **x** or **Esc** to close the monitor and resume CPUs that were running when it was entered.

The monitor works with all six CPU types (IE64, IE32, M68K, Z80, 6502, X86) and handles multi-CPU scenarios, including coprocessors. It is also exposed to IEScript Lua via the `dbg.*` API for scripted debugging workflows. See iescript.md for the full `dbg.*` module reference.

Availability: The monitor is part of the engine - no command-line flag is required. The F9 overlay is available when a video backend attaches the monitor; headless builds still use the same monitor core for tests and scripted debugging. If an IEScript Lua REPL is also bound (F8), the monitor takes priority while it is active and F8 is suppressed.

Quick Start

1. Run a target program: `./bin/IntuitionEngine program.ie64`
2. Press **F9** to freeze and enter the monitor
3. Type `r` to see registers
4. Type `d` to disassemble around the program counter
5. Type `s` to single-step one instruction
6. Type `x` to resume execution

Address Formats

Address arguments accept these formats:

Format	Example	Description
`$hex`	`$1000`	Hex with dollar sign (classic monitor convention)
`0xhex`	`0x1000`	Hex with 0x prefix
bare hex	`1000`	Bare hexadecimal (default)
`#decimal`	`#4096`	Decimal with hash prefix

Expression Evaluation

Several address arguments support simple expressions with register names and arithmetic:

```
> d pc+$20          (disassemble from PC+0x20)
> m sp-8            (memory dump from SP-8)
> d $1000+r1        (disassemble from 0x1000 + R1)
> b pc+$100         (set breakpoint at PC+0x100)
```

Operators: `+` and `-` only. Each term is either a register name or a numeric address.

Expression support is command-specific. It is available for `d`, `list`, `m`, `b`, `g`, `save`, the destination address in `load`, `u`, `ww`, `wc`, `sym add`, `sym resolve`, the optional base in `sym loadlbl`, `addr`, `pg add`, `who`, `trace watch`, `trace history`, and `e`. The low-level byte/range commands `w`, `f`, `h`, `c`, `t`, and `bc` parse literal addresses only. The IE64 assembler command `A` also parses a checked unsigned literal physical address only.

Symbols loaded with `sym add`, `sym loadlbl`, or `sym loadelf` can be used as expression terms, for example `b main+0x10`. Symbols are scoped per CPU name.

`d /s [addr] [count]` asks the monitor to interleave source locations from DWARF line data when available. `list [addr]` prints the nearest source location, or a no-source message for CPUs/builds without line information.

Scripted equivalents: `sym.load_dwarf(path)` and `dbg.source_at(addr)`.

Argument Parsing Matrix

Command or argument group	Expression support	Literal-only parts
`d`, `list`, `m`, `b`, `g`, `u`, `ww`, `wc`, `addr`, `who`, `trace watch`, `trace history`, `e`	Address argument accepts register, symbol, `+`, and `-` terms	Counts, widths, and non-address switches remain command-specific literals
`save <start> <end> <file>`	Start and end address operands both accept register, symbol, `+`, and `-` terms	Filename is a host path argument, not an expression
`load`	Destination address accepts expressions	Filename is a host path argument, not an expression
`sym add`, `sym resolve`, `sym loadlbl`	`sym add` address, `sym resolve` address, and `sym loadlbl` optional base accept register, symbol, `+`, and `-` terms	Symbol names, symbol kinds, and filenames are not expressions
`pg add <start> <end> <rwx> [cpu=...]`	Start and end operands accept register, symbol, `+`, and `-` terms	Permission string and CPU scope are parsed by the page-guard grammar
`w`, `f`, `h`, `c`, `t`, `bc`	None	Low-level byte/range addresses are parsed as literal addresses
`A`	None	IE64 assembly address is a checked unsigned physical-address literal

The matrix is intentionally command-facing: condition expressions for breakpoints, reverse timeline queries, page guards, and access filters use their own grammar and are documented in the relevant command sections below.

Conditional Breakpoints

Breakpoints accept either the legacy single comparison form or an `if` expression:

```
> b $2000 R1==$10
> b $2000 if R1==$10 && (hitcount>2 || b($3000)!=$00)
```

Form	Meaning
`REG==$10`	Register comparison
`hitcount>5`	Breakpoint hit-count comparison

Form	Meaning
`b($1000)`	8-bit memory read
`w($1000)`	16-bit memory read
`l($1000)`	32-bit memory read
`q($1000)`	64-bit memory read
`&&`, `||`, `(...)`	Boolean composition

CPU	Register names
IE64	`R0`-`R31`, `SP`, `PC`
IE32	IE32 general register names, `SP`, `PC`
M68K	`D0`-`D7`, `A0`-`A7`, `SP`, `PC`, status registers exposed by the adapter
Z80	`A`, `F`, `B`, `C`, `D`, `E`, `H`, `L`, shadow/index registers, `SP`, `PC`
6502	`A`, `X`, `Y`, `SP`, `PC`, `SR`
X86	x86 adapter register names including `EAX`-`EDI`, `ESP`, `EBP`, `EIP`

Trace Ring

`tracering on [size]` enables a per-CPU rolling instruction ring. `show [n]` dumps the last entries for the focussed CPU.

Scripted equivalents: `dbg.tracering_on([size])`, `dbg.tracering_off()`, and `dbg.tracering_show([n])`.

Command	Description
`tracering on`	Enable the default 4096-entry ring
`tracering on 256`	Enable and resize the focussed CPU ring
`tracering off`	Stop recording
`show 32`	Show the last 32 recorded instructions

CPU	Entry shape
IE64	64-bit PC, raw bytes, disassembly
IE32	32-bit PC, raw bytes, disassembly
M68K	32-bit PC, raw bytes, disassembly
Z80	16-bit PC, raw bytes, disassembly
6502	16-bit PC, raw bytes, disassembly
X86	32-bit PC, raw bytes, disassembly

Reverse Step

`rs` is an alias for the existing `bs` backstep command. It restores the focussed CPU from the CPU-local step-history snapshot and does not rewind other CPUs, devices, audio, video, timers, DMA, or MMIO side effects.

Whole-Machine Reverse

`rg` targets the latest retained whole-machine snapshot, including snapshots captured when the monitor stops for a breakpoint, watchpoint, guard, break-in, or fault. When a retained predecessor exists, IEMon restores that predecessor and deterministically re-executes to the target boundary; when the target is the oldest retained state, it restores it directly. `rt <expr>` walks backwards through retained whole-machine snapshots and uses the same replay path for the newest snapshot where the focussed CPU satisfies the breakpoint-expression syntax. `tl back` is a timeline-view shorthand for `rg`. `history horizon` reports the retained reverse snapshot horizon, checkpoint count, delta count, and approximate retained bytes. `history config` prints the current snapshot-chain settings; `history config <delta-interval> <delta-miB> <checkpoints> [snapshots]` changes them and can be placed in a trusted `.iemonrc`. `tl [count]` shows the merged timeline from access events, instruction trace entries, and monitor stop events using the shared sequence assigned when each event is recorded; stop events include `snap=N` when they captured a reverse boundary.

Whole-machine snapshots cover all monitor-registered CPUs, shared bus RAM, sparse IE64 backing memory, and registered versioned device blobs. The history stores full sparse checkpoints plus sparse deltas anchored to a retained checkpoint; `rg` and `rt` materialise deltas before replay or restore. The production monitor registers the main video chip, sound chip, terminal MMIO, command-style host helpers (file I/O, media loader, program executor, and coprocessor manager), compatibility audio/video engines (PSG/AY, SN76489, SID/SID2/SID3, TED audio, POKEY, VGA, ULA, TED video, ANTIC/GTIA, and Voodoo), and optional host bridges when present. Additional devices join the same contract through `RegisterSnapshotDevice`. Timeline replay depends on deterministic device snapshot and restore behaviour.

Scripted equivalents: `dbg.history_horizon()`, `dbg.history_config([opts])`, `dbg.device_list()`, `dbg.device_snapshot(name)`, and `dbg.device_diff(a,b)`.

Project RC Files

IEMon can load project-local `.iemonrc` files after they have been explicitly trusted. `rc list` searches from the current directory up to the file system root and prints each candidate with its SHA-256 hash and trust state. `rc trust [file]` records the current absolute path and hash in the IEMon trust store, and `rc load [file]` runs the file only while the stored hash still matches. Trusted rc files are auto-loaded once per matching hash when the monitor has exactly one registered CPU; multi-CPU monitor sessions require explicit `rc load` to avoid applying setup to the wrong focus.

RC files are deliberately limited to debugger setup commands: `b`, `bc`, `ww`, `wc`, `bpm*`, `pg add|clear|list`, `sym add`, `history config`, `layout`, and safe `alias` definitions whose target command is also allowed. Commands that load host files, run scripts, continue execution, enter assemble mode, or modify guest memory are rejected.

Command	Description
`rc list`	Show discovered `.iemonrc` files and trust state
`rc trust .iemonrc`	Trust the current contents of a project rc file
`rc load .iemonrc`	Load a trusted rc file

Command History

Command history persists under the IEMon home directory, normally `~/.iemon/history`. Up and Down browse entries, and Ctrl-R searches backwards using the current input text as the query. Press Ctrl-R again to continue to the next older match.

Watchpoints

Legacy `ww <addr>` sets a one-byte write watchpoint. SoftICE-style `bpm*` commands add read/write mode and width syntax:

Command family	Mode	Width
`bpmbr`, `bpmrb`	Read	Byte
`bpmbw`, `bpmwb`	Write	Byte
`bpmb`, `bpmba`, `bpmab`	Read/write	Byte
`bpmwr`, `bpmrw`	Read	Word
`bpmww`	Write	Word
`bpmw`, `bpmwa`, `bpmaw`	Read/write	Word
`bpmdr`, `bpmrd`	Read	Long
`bpmdw`, `bpmwd`	Write	Long
`bpmd`, `bpmda`, `bpmad`	Read/write	Long
`bpmqr`, `bpmrq`	Read	Quad
`bpmqw`, `bpmwq`	Write	Quad
`bpmq`, `bpmqa`, `bpmaq`	Read/write	Quad

Read and read/write watchpoints are backed by CPU access-site instrumentation. Write-only `ww` compatibility remains available.

Page Guards

`pg` manages debug access-service guards:

Command	Description
`pg add $1000 $10ff r`	Break on reads in a range
`pg add $1000 $10ff rwx cpu=current`	Scope to the focussed CPU
`pg list`	List guards
`pg clear`	Clear guards

The shared debug access service stores guard policy and emits monitor events with CPU id and access kind. Bus-mediated `Read8/16/32` and `Write8/16/32` paths are instrumented. CPU-local direct memory paths and instruction fetch hooks must route through the access service for guard and history coverage.

Access History

The access service can retain a bounded event log for read/write/fetch hooks. Data read/write paths for all six CPU families are attributed to the active CPU for guard/break semantics. Bus activity that is not attributed by a CPU context is recorded with `cpu=-1` for history only; it does not trip guards because monitor breaks must focus a real CPU.

Command	Description
`accesslog on 1024`	Keep the last 1024 access events
`accesslog show 16`	Show recent events
`accesslog off`	Disable and clear the access log
`who wrote $4000`	Show the last recorded write covering an address
`who read $4000`	Show the last recorded read covering an address
`who fetched $4000`	Show the last recorded execute/fetch covering an address
`bfirst write mmio`	Break once on the first write to a named region
`trace mmio mmio 16`	Show recent access events inside a named region

Wide accesses are matched by range, so `who wrote $4003` reports a prior 4-byte write at `$4000`. Like page guards, this depends on CPU and bus access sites calling the shared debug access service.

Count arguments for `s`, `m`, `trace`, and `bt` are decimal by default. Use `$` or `0x` for hexadecimal counts. The `d` (disassemble) line count is the exception: it is parsed with the address parser, so a bare count is hexadecimal and `#decimal` is accepted (for example, `d $2000 10` disassembles 0x10 = 16 lines, while `d $2000 #10` disassembles 10 lines). Bare address arguments remain hexadecimal by default. The `#decimal` prefix is recognised for address/value arguments and for the `d` line count, but not for other count arguments.

Arguments containing spaces can be wrapped in double quotes, for example `save $1000 $1FFF "my dump.bin"`. Inside double quotes, a backslash escapes the following character.

Command Reference

Command Surface

The command surface below reflects the monitor command registry plus dispatch-level aliases. Detailed sections later in this chapter expand the commands that need longer notes.

Command	Purpose
`r`	Show or change registers
`a` / `A`	Enter IE64 one-instruction assemble mode
`d`	Disassemble memory; `/s` shows source lines when available
`list`	Show source location for an address
`m`	Dump memory as hex and ASCII
`s`	Single-step the focussed CPU
`bs` / `rs`	Step the focussed CPU backwards using CPU-local history
`rg`	Replay or restore to the previous whole-machine reverse boundary
`rt`	Replay or restore to the latest reverse boundary matching an expression
`tl`	Show the merged timeline or scrub backwards
`g`	Continue execution, optionally from a new PC

Command	Purpose
`u`	Run until an address is reached
`x`	Close the monitor and resume CPUs that were running
`b`	Set a breakpoint with an optional condition
`bc`	Clear one breakpoint or all breakpoints
`bl`	List breakpoints
`ww`	Set a legacy one-byte write watchpoint
`wr` / `wrw`	Set a legacy one-byte read/write watchpoint
`bpm`	Set read/write watchpoints by access mode and width
`bpmbr` / `bpmrb`	Set a byte read watchpoint
`bpmbw` / `bpmwb`	Set a byte write watchpoint
`bpmb` / `bpmba` / `bpmab`	Set a byte read/write watchpoint
`bpmwr` / `bpmrw`	Set a word read watchpoint
`bpmww`	Set a word write watchpoint
`bpmw` / `bpmwa` / `bpmaw`	Set a word read/write watchpoint
`bpmdr` / `bpmrd`	Set a long read watchpoint
`bpmdw` / `bpmwd`	Set a long write watchpoint
`bpmd` / `bpmda` / `bpmad`	Set a long read/write watchpoint
`bpmqr` / `bpmrq`	Set a quad read watchpoint
`bpmqw` / `bpmwq`	Set a quad write watchpoint
`bpmq` / `bpmqa` / `bpmaq`	Set a quad read/write watchpoint
`wc`	Clear one watchpoint or all watchpoints
`wl`	List watchpoints
`bt`	Show a symbol-aware stack backtrace
`sym`	Manage symbols for the focussed CPU
`map`	List the memory map for the focussed CPU
`addr`	Describe the memory region containing an address
`pg`	Add, list, or clear page-access guards
`accesslog`	Record read/write/fetch access events
`who`	Find the last reader, writer, or fetcher of an address
`bfirst`	Break once on the first access to a named region
`trace`	Trace instructions, files, write history, or MMIO access events
`history`	Show or tune reverse-debugging snapshot history
`tracering`	Enable or disable the per-CPU instruction trace ring
`show`	Show the tail of the instruction trace ring

Command	Purpose
`fault`	Break before selected guest fault handlers run
`cpu`	List CPUs, change focus, or manage coprocessor worker slots
`freeze`	Freeze a CPU or all CPUs
`thaw`	Resume a frozen CPU or all CPUs
`layout`	Render a named monitor view preset
`alias`	Create or list command aliases
`rc`	List, trust, or load project-local IEMon rc files
`bug`	Print a copyable debugger report bundle
`io`	Show I/O registers
`e`	Enter hex editor mode
`f`	Fill memory
`w`	Write bytes to memory
`h`	Hunt for a byte pattern
`c`	Compare two memory ranges
`t`	Transfer memory
`save`	Save memory to a host file
`load`	Load a host file into memory
`ss`	Save a CPU-local state snapshot
`sl`	Load a CPU-local state snapshot
`fa`	Freeze audio output
`ta`	Thaw audio output
`script`	Run a monitor command script
`macro`	Define a semicolon-separated command macro
`?` / `help`	Show command help

Registry Syntax Inventory

This compact syntax inventory mirrors the command help registry. Detailed sections below describe argument meaning, output, errors, and side effects.

- `r`
- `r <name> <value>`
- `A <addr>`
- `a <addr>`
- `d [/s] [addr] [count]`
- `list [addr]`
- `m [addr] [lines]`
- `s [count]`

- bs
- rs
- rg
- rt <expr>
- tl [count]
- tl back
- g [addr]
- u <addr>
- x
- b <addr> [if <expr>]
- b <addr> <legacy-condition>
- bc <addr|*>
- bl
- ww <addr>
- bpmbr|bpmbw|bpmb <addr>
- bpmwr|bpmww|bpmw <addr>
- bpmdr|bpmdw|bpmd <addr>
- bpmqr|bpmqw|bpmq <addr>
- wc <addr|*>
- wl
- bt [depth]
- sym add <name> <addr> [func|object|label]
- sym loadlbl <file> [base]
- sym loadelf <file>
- sym lookup|resolve|list ...
- map
- addr <addr>
- pg add <start> <end> <rwx> [cpu=all|current]
- pg list
- pg clear
- accesslog on [size]
- accesslog off
- accesslog show [count]
- who read|wrote|fetched <addr>
- bfirst read|write|fetch <region-name>
- trace <count>
- trace file <path|off>
- trace watch add|del|list <addr>
- trace history show|clear <addr|*>
- trace mmio <region> [count]
- history horizon
- history config [delta-interval] [delta-miB] [checkpoints] [snapshots]
- tracering on|off [size]
- show [count]
- fault on|off|list
- fault break <kind>
- fault clear <kind>
- cpu

- `cpu <id|label>`
- `cpu online [--replace] <type|path.ie*> [path.ie*]`
- `cpu offline <id|label|type>`
- `freeze <id|label|*>`
- `thaw <id|label|*>`
- `layout cpu|trace|debug`
- `layout list`
- `layout save <name>`
- `alias`
- `alias <name> <command...>`
- `rc list`
- `rc trust [file]`
- `rc load [file]`
- `bug [trace-count]`
- `io [device|all]`
- `e <addr>`
- `f <start> <end> <byte>`
- `w <addr> <bytes..>`
- `h <start> <end> <bytes..>`
- `c <start> <end> <dest>`
- `t <start> <end> <dest>`
- `save <start> <end> <file>`
- `load <file> <addr>`
- `ss [file]`
- `sl [file]`
- `fa`
- `ta`
- `script <file>`
- `macro <name> <cmds..>`

Command Effect Matrix

Commands fall into four operational classes. This matrix is intended for scripts, transcripts, and debugger sessions where the difference between inspection and state mutation matters.

Class	Commands	Effect
Inspection only	`r`, `d`, `list`, `m`, `bl`, `wl`, `bt`, `map`, `addr`, `pg list`, `accesslog show`, `who`, `history horizon`, `show`, `io`, `layout`, `alias`, `rc list`, `bug`, `?`, `help`	Read monitor, CPU, memory, trace, or device state and append output.
CPU execution control	`s`, `g`, `u`, `x`, `bs`, `rg`, `rt`, `freeze`, `thaw`, `cpu`	Step, resume, stop, reverse, change focus, or change worker lifecycle. These commands can change PC, CPU running state, reverse-history position, or focussed CPU.
Memory and debugger mutation	`r <name> <value>`, `w`, `f`, `t`, `load`, `e`, `b`, `bc`, `ww`, `bpm*`, `wc`, `pg add`, `pg clear`, `accesslog on`, `accesslog off`, `bfirst`, `trace watch`, `trace history clear`, `tracering`, `fault`, `sym`, `ss`, `sl`	Modify guest memory, register values, monitor break/watch state, trace settings, page guards, symbol tables, or CPU-local snapshot state.

Class	Commands	Effect
Host file or session mutation	`save`, `trace file`, `script`, `macro`, `alias <name>`, `layout save`, `rc trust`, `rc load`, `fa`, `ta`	Read or write host files, execute monitor command files, define session helpers, trust project rc files, or change host audio output state.

State-changing commands are intentionally not hidden behind confirmation in the monitor command line. When the same operation is exposed through IEScript debug helpers, the script sandbox and debugger command filter may impose additional path or command restrictions.

Execution Control

Break-In

Host-side break-in requests stop the focussed execution path through the same event pipeline as breakpoints and enter IEMon with a `BREAK-IN` message. Debug adapters expose `RequestBreakIn`, `BreakInRequested`, and `ConsumeBreakIn` so CPU dispatch loops can poll at instruction boundaries. Display builds still use F9 for the overlay toggle; TTY-style hosts can bind the SoftICE-style Ctrl-D byte to `RequestBreakIn()` through the hotkey listener.

CPU	Break-in boundary
IE64	Before the next instruction observed by the debug adapter
IE32	Before the next instruction observed by the debug adapter
M68K	Before the next 68020 instruction observed by the debug adapter
Z80	Before the next opcode observed by the debug adapter
6502	Before the next opcode observed by the debug adapter
X86	Before the next instruction observed by the debug adapter

Memory Map

The `map` command lists named memory regions for the focussed CPU. The `addr <addr>` command resolves one address to a region and accepts the same symbol-aware address expressions as `d` and `b`. The regions are views of the shared MachineBus memory map; CPU adapters may translate addresses, but they do not create private RAM.

CPU	Region divergence
IE64	Standard shared machine RAM, stack, VRAM, and `0xF0000-0xFFFFF` MMIO regions
IE32	Standard shared machine RAM, stack, VRAM, and `0xF0000-0xFFFFF` MMIO regions
M68K	Standard shared machine RAM, stack, VRAM, and `0xF0000-0xFFFFF` MMIO regions
X86	Standard regions plus x86 runner bank windows translated by the bus adapter
Z80	`0xF000-0xF0FF` direct MMIO window and `0xA0-0xAD` VGA port range
6502	Page-1 stack, `0xF000-0xF0FF` direct MMIO, VGA at `0xD700-0xD70D`, and ULA at `0xD800-0xD817`

`s [count]` - Single-Step

Execute one (or more) instructions on the focussed CPU. Displays changed registers in green, followed by the next instruction to be executed.

```
> s
Step: 1 instruction(s), 1 cycle(s)
  R1: $0 -> $2A
> 001008: E0 00 00 00 00 00 00 00  nop
```

Step 10 instructions (counts are decimal):

```
> s 10
Step: 10 instruction(s), 10 cycle(s)
```

Single-step uses frozen-debug execution semantics. `WAIT` instructions count as one stepped instruction and do not consume their requested real-time delay while the monitor is active; continued execution with `g` uses normal processor wait timing.

`g [addr]` - Go/Continue

Resume execution and exit the monitor. Optionally set the PC before resuming.

```
> g          (resume from current PC)
> g $2000    (set PC to $2000, then resume)
```

If the supplied address fails to parse, the command resumes from the current PC without setting it and without an error message. If it parses but is out of range for the focussed CPU, the command prints a red `ValidateAddress` error and stays in the monitor instead of resuming.

`x` - Exit Monitor

Close the monitor overlay. CPUs that were running when the monitor was entered are resumed; CPUs that were already frozen stay frozen. Equivalent to pressing Esc.

Inspection

`r` - Show Registers

Display all registers of the focussed CPU. Registers that changed since the monitor's last saved register snapshot are shown in green. The snapshot is refreshed when the monitor is entered, after `s`, after `bs`, after `sl`, and after changing CPU focus.

```
> r
PC   $0000000000001000
R0   $0000000000000000
R1   $000000000000002A    (green = changed)
...
```

`r <name> <value>` - Set Register

Modify a register value. Values are truncated by the CPU adapter to the target register width where applicable.

```
> r pc $2000
PC = $2000
> r r1 #42
R1 = $2A
```

`d [addr] [count]` - Disassemble

Disassemble instructions starting from an address (default: current PC, 16 lines). The current PC is marked with `>`, breakpoints with `*`, and branch targets within the visible window with `T`.

```
> d
> 001000: 01 81 00 00 2A 00 00 00  move.l r1, #$2A
  001008: E0 00 00 00 00 00 00 00  nop
* 001010: 01 81 00 00 FF 00 00 00  move.l r1, #$FF

> d $2000 8    (disassemble 8 instructions from $2000)
```

Branch annotations: Backward branch and jump instructions with a known target are marked with `<- LOOP` in magenta. Lines that are branch targets within the visible window are prefixed with `T`.

If an address or count argument does not parse, `d` keeps the default for that argument and does not print an error.

`m [addr] [count]` - Memory Dump

Display memory in hex + ASCII format (default: from PC, 8 lines of 16 bytes). The `count` argument is a line count, not a byte count. The address column follows the focussed CPU width, so IE64 dumps preserve full 64-bit addresses.

```
> m $1000 4
001000: 01 81 00 00 2A 00 00 00  E0 00 00 00 00 00 00 00  ....*...........
001010: 01 81 00 00 FF 00 00 00  00 00 00 00 00 00 00 00  ................
001020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
001030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Memory Modification

`w <addr> <bytes..>` - Write Bytes

Write individual bytes to memory. Byte values are parsed as monitor values and then truncated to 8 bits.

```
> w $1000 DE AD BE EF
Wrote 4 byte(s) at $1000
```

`A <addr>` - IE64 Assemble Mode

Enter IE64-only one-instruction assemble mode at a checked unsigned physical address. The prompt changes to the current write address:

```
> A $1000
IE64 assemble at $0000000000001000; empty line exits
asm $0000000000001000> move.l r2,#42
$0000000000001000: 01 15 00 00 2A 00 00 00  move.l r2, #$2A
asm $0000000000001008>
```

Each non-empty line must assemble to exactly one IE64 instruction. A successful line writes 8 bytes, prints the address, bytes, and disassembly, advances the prompt by 8, and flushes the full IE64 JIT/code cache. A failed line leaves memory and the current address unchanged. Press Enter on an empty line to exit; empty assemble-mode input is not repeated through command history.

The write path is physical RAM-only. MMIO ranges, unmapped addresses, and addresses outside the backing RAM are rejected instead of invoking device side effects. The command is available only when the focussed CPU is IE64.

Monitor assemble mode intentionally excludes source-file features: labels, directives, `include`, `incbin`, output-format controls, files, and multi-instruction pseudo-ops are rejected. Pseudo-ops that still encode as one instruction, such as `la`, may assemble, but `li` is rejected because it expands to more than one instruction. The one-instruction assembler accepts the same public IE64 mnemonics as the single-instruction backend used by the monitor, including the FP64 transcendental opcodes `dsin`, `dcos`, `dtan`, `datan`, `dlog`, `dexp`, and `dpow`. The standalone `ie64asm` CLI remains the full source-file assembler.

Assemble mode is interactive only. Monitor scripts, macros, project `.iemonrc` files, and IEScript raw monitor wrappers such as `dbg.command`, `dbg.command_output`, `dbg.run_script`, and `dbg.macro` cannot enter or feed it.

`f <start> <end> <byte>` - Fill Memory

Fill a memory range with a single byte value.

```
> f $2000 $20FF 00
Filled $2000-$20FF with $00
```

The fill range is capped at 1 MiB (`0x100000` bytes); larger ranges are rejected with `Invalid range`. The fill value is truncated to 8 bits.

Memory Export/Import

`save <start> <end> <filename>` - Save Memory to File

Save a range of memory to a raw binary file. The maximum requested save range is the bus-reported total guest RAM when the bus is available, falling back to 32 MiB when it is not. Ranges larger than the cap are rejected with `Range too large (max N bytes)`. The command reads through the focussed CPU adapter; use an address range that is meaningful for that CPU. Adapter short reads are not reported separately.

```
> save $1000 $1FFF dump.bin
Saved 4096 bytes ($1000-$1FFF) to dump.bin
```

`load <filename> <addr>` - Load File into Memory

Load a raw binary file into memory at the specified address.

```
> load dump.bin $2000
Loaded 4096 bytes from dump.bin to $2000
```

File size is capped for safety at 32 MiB. Files larger than that are rejected with `File too large (max 32MB)`. The destination address is not validated after parsing. Adapter behaviour near the top of the address space is CPU-specific: 6502 and Z80 writes wrap through 16-bit addresses; x86 and some M68K paths write through a 32-bit bus address; IE32 rejects writes that do not fit in its backing memory; IE64 may write through the machine bus for high addresses. Pick the destination address with that in mind.

Memory Tools

`h <start> <end> <bytes..>` - Hunt/Search

Search a memory range for a byte pattern. Pattern values are truncated to 8 bits.

```
> h $0 $FFFF DE AD
Found at $1000
Found at $3456
```

Hit reporting is capped at 256 matches. When the cap is reached, the search prints `... (truncated)` and stops. Zero matches print `Not found` in dim text.

`c <start> <end> <dest>` - Compare Memory

Compare two memory ranges and report differences. If the requested range is outside the focussed CPU's readable memory, the comparison is limited by what the adapter returns. If both reads return no comparable bytes, the command prints `Identical`.

```
> c $1000 $100F $2000
$1000: DE != 00 (at $2000)
$1001: AD != 00 (at $2001)
```

Diff output is capped at 256 mismatches. When the cap is reached, comparison prints `... (truncated)` and stops. Identical compared bytes print `Identical` in green.

`t <start> <end> <dest>` - Transfer/Copy Memory

Copy a memory range to another location. The command reads through the focussed CPU adapter and writes the returned bytes to the destination; overlapping copies are read before writing. The success message reports the requested byte count.

```
> t $1000 $100F $2000
Transferred 16 bytes from $1000 to $2000
```

`u <addr>` - Run Until

Run the program until the PC reaches the specified address, then stop and re-enter the monitor. Internally sets a temporary breakpoint that is automatically cleared when hit.

```
> u $2000
```

The monitor exits and execution resumes. When the PC reaches `$2000`, the monitor activates automatically and the temporary breakpoint is removed. If run-until temporarily disables an existing conditional breakpoint, the condition is restored when that stop is handled.

If `u` creates a new temporary breakpoint and execution never reaches the target address before the monitor is re-entered for another reason, that temporary breakpoint remains set and will fire on a future run. Use `bc <addr>` to clear it explicitly if you no longer want the stop.

`bs` - Backstep (Undo Step)

Rewind the focussed CPU to the state before the last `s` (single-step) command. Restores focussed CPU registers and the CPU-local memory snapshot captured by that adapter.

```
> s
Step: 1 instruction(s), 1 cycle(s)
  R1: $0 -> $2A
> bs
Backstep: restored to PC=$1000 (CPU+memory)
```

A ring buffer of up to 32 CPU-local snapshots is maintained. Each stepped instruction saves a snapshot before stepping; `bs` pops the most recent one.

Note: Only the focussed CPU adapter's registers and the captured memory span are restored. The captured span starts at address 0 and is 64 KiB for 16-bit adapters or 32 MiB for wider adapters. Device/chip runtime state (timers, audio envelopes, video scanline position), other CPUs, and coprocessor state are not included.

Breakpoints

`b <addr> [condition]` - Set Breakpoint

Set a breakpoint at an address. When normal execution reaches this PC, the monitor activates before that instruction is executed. An optional condition causes the breakpoint to fire only when the condition is true.

```
> b $1010
Breakpoint set at $1010

> b $1010 r1==$FF
Breakpoint set at $1010 if R1==$FF

> b $2000 [$1000]==$42
Breakpoint set at $2000 if [$1000]==$42

> b $3000 hitcount>#10
Breakpoint set at $3000 if hitcount>$A
```

Condition syntax:

Format	Description
`r1==$FF`	Register R1 equals 0xFF
`[$1000]==$42`	Memory byte at $1000 equals 0x42
`[$1000].W==$1234`	Memory word at $1000 equals 0x1234
`[$1000].L==$12345678`	Memory long at $1000 equals 0x12345678
`hitcount>#10`	Breakpoint hit count exceeds decimal 10

Operators: ==, !=, <, >, <=, >=

Legacy memory conditions use `[$addr]`, `[$addr].W`, or `[$addr].L`. The `if <expr>` form also accepts `b(addr)`, `w(addr)`, `l(addr)`, and `q(addr)` memory terms for byte, word, long, and quad reads. Condition values use the normal address/value syntax (`$hex`, `0xhex`, bare hex, or `#decimal`). Memory condition addresses inside both syntaxes are literals, not register or symbol expressions. `.W`/`w(addr)`, `.L`/`l(addr)`, and `q(addr)` memory values use the focussed CPU's byte order: M68K is big-endian; the other current CPU adapters are little-endian.

`bc <addr>` / `bc *` - Clear Breakpoint(s)

Clear a single breakpoint by address, or clear all breakpoints on the currently focussed CPU. `bc *` clears only the focussed CPU's breakpoints; use `cpu <id>` then `bc *` on each CPU to clear globally.

```
> bc $1010
Breakpoint cleared at $1010

> bc *
All breakpoints cleared
```

`bl` - List Breakpoints

List all breakpoints across all CPUs, including conditions and hit counts.

```
> bl
$1010 (id:0 IE64)
$2000 if R1==$FF (hits:3) (id:0 IE64)
$400 (id:3 coproc:Z80)
```

CPUs with no breakpoints are skipped silently; an empty list produces no output.

When a breakpoint is hit during normal execution, the monitor activates automatically, freezes running CPUs, and focuses on the CPU that hit the breakpoint:

```
BREAK at $1010 on IE64 (id:0)
```

Watchpoints

`ww <addr>` / `bpm* <addr>` - Set Watchpoint

Monitor a memory address for reads, writes, or either mode. Width-aware `bpm*` commands support byte, word, long, and quad ranges and trigger on overlapping accesses.

```
> ww $1000
W1 watchpoint set at $1000

> bpmdr $2000
R4 watchpoint set at $2000

> bpmq $3000
RW8 watchpoint set at $3000
```

When triggered:

```
WATCH $1000: $00 -> $FF at PC=$1234 on IE64 (id:0)
```

`wc <addr>` / `wc *` - Clear Watchpoint(s)

Clear a single watchpoint by address, or clear all watchpoints on the focussed CPU.

```
> wc $1000
Watchpoint cleared at $1000

> wc *
All watchpoints cleared
```

`wl` - List Watchpoints

List all watchpoints across all CPUs.

```
> wl
W $1000 (id:0 IE64)
W $2000 (id:3 coproc:Z80)
```

Multi-CPU Commands

`cpu` - List CPUs

List all registered CPUs with their ID, label, status, and program counter. When a coprocessor manager is attached, offline coprocessor worker slots are also shown. The focussed CPU is marked with `*`.

```
> cpu
*id:0   IE64         [FROZEN ]  PC=$1000
 id:1   coproc:Z80   [FROZEN ]  PC=$40
 id:2   coproc:6502  [FROZEN ]  PC=$200
 id:-   coproc:IE32  [OFFLINE]  PC=-
```

Offline rows are worker slots only. They are not general CPU hot-plug targets and do not include the primary boot CPU.

`cpu <id|label>` - Switch Focus

Switch the focussed CPU by stable ID or label. All register/disassembly/step commands operate on the focussed CPU.

```
> cpu 1
Focussed on id:1 coproc:Z80
```

Labels are matched exactly (case-insensitive).

If an exact label matches multiple CPUs, the command lists matches and asks for the ID:

```
> cpu coproc:z80
Ambiguous label, use ID:
  id:1 coproc:Z80
  id:5 coproc:Z80
```

`cpu online` - Start Coprocessor Worker

Start an offline coprocessor worker slot through the same isolated worker lifecycle used by `COPROC_CMD_START`.

Scripted equivalents: `dbg.cpu_online(type_or_path [, path_or_replace] [, replace])` and `dbg.cpu_offline(id_or_label)`.

```
> cpu online z80
Online z80 as coproc:Z80

> cpu online svc.ie80
Online z80 as coproc:Z80

> cpu online z80 svc.ie80
Online z80 as coproc:Z80

> cpu online --replace svc.ie80
Online z80 as coproc:Z80
```

`cpu online <type>` uses the currently staged/default coprocessor service image. The staging path is the same one used by `-coproc-svc` / `-coproc` and `COPROC_NAME_PTR`; if no service path is staged, the command fails without starting a worker. Path validation is performed by the coprocessor manager, so the monitor does not bypass the worker loader's sandbox.

`cpu online <path.ie*>` infers the worker CPU from the typed image extension. `cpu online <type> <path.ie*>` requires the type and extension to match. Recognised extensions are `.ie64` (IE64), `.iex`/`.ie32` (IE32), `.ie68` (M68K), `.ie80` (Z80), `.ie65` (6502), and `.ie86` (x86).

Paths are resolved by the coprocessor manager under its configured base directory. Absolute paths, `..`, unsupported extensions, missing files, oversized worker images, and type/extension mismatches are rejected. Starting an already-online worker is rejected unless `--replace` is supplied.

`cpu offline <id|label|type>` - Stop Coprocessor Worker

Stop an online coprocessor worker and unregister it from IEMon.

```
> cpu offline z80
Offline coproc:Z80

> cpu offline coproc:z80
Offline coproc:Z80
```

Only coprocessor worker slots can be offlined. Primary CPUs and other registered monitor adapters are rejected.

`freeze <id|label|*>` - Freeze CPU

Freeze a specific CPU or all CPUs.

```
> freeze 1       (freeze CPU id:1)
> freeze coproc:z80     (freeze by label, must be unambiguous)
> freeze *       (freeze all)
```

`thaw <id|label|*>` - Thaw CPU

Resume a specific CPU while the monitor stays open. This allows advanced debugging where some CPUs run while others are frozen.

```
> thaw 1         (thaw CPU id:1)
> thaw *         (thaw all)
```

Stack Trace

`bt [depth]` - Backtrace

Walk the stack and display return addresses. Default depth is 16. If symbols are loaded, `bt` resolves frame labels and filters obvious stack noise by requiring a symbol hit and rejecting addresses that fall inside known stack, VRAM, or MMIO regions.

```
> bt
#0 $001234
#1 $005678
#2 $009ABC

> bt 4
#0 $001234
#1 $005678
#2 $009ABC
#3 $00DEF0
```

Stack walking is CPU-specific and best-effort. A missing frame does not prove there was no caller; it means IEMon could not identify a plausible saved return address through the adapter and symbol filters.

CPU	Source	Slot Size	Notes
IE64	SP	8 bytes (LE)	Full 64-bit return addresses
IE32	SP	4 bytes (LE)	-
M68K	A6 frame-link chain, then A7 scan	4 bytes (BE)	Walks `prevA6 = mem[A6]; ret = mem[A6+4]` for LINK/UNLK frames. If A6 is invalid, falls back to an SP scan and lets symbol filtering reject noise
Z80	SP	2 bytes (LE)	-
6502	SP (page 1)	2 bytes (LE)	Each frame is tagged `(low confidence)` in output; reads from `$0100 + ((SP+1) & 0xFF)` upward, adding +1 because JSR pushes return-1
X86	EBP chain, then ESP scan	4 bytes (LE)	Uses an EBP frame chain when it looks valid, otherwise scans ESP

CPU-Local Snapshot Save/Load

`ss [filename]` - Save State

Save a snapshot of the focussed CPU's registers and a fixed CPU memory span to disk. Default filename: `snapshot.iem`.

```
> ss
State saved to snapshot.iem (CPU+memory)

> ss mystate.iem
State saved to mystate.iem (CPU+memory)
```

`sl [filename]` - Load State

Restore a previously saved snapshot, overwriting the focussed CPU registers and the memory span stored in the snapshot.

```
> sl
State loaded from snapshot.iem (CPU+memory)

> sl mystate.iem
State loaded from mystate.iem (CPU+memory)
```

Note: `ss`/`sl` operate only on the focussed CPU. Current snapshots capture memory starting at address 0: 64 KiB for 16-bit adapters and 32 MiB for wider adapters. Snapshot files gzip-compress that memory on disk. Other CPUs and device/chip runtime state (timers, audio envelopes, video scanline position) are not included. `sl` refuses to load a snapshot whose CPU type differs from the focussed CPU.

Trace and Write History

`trace <count>` - Trace Instructions

Execute up to N instructions on the focussed CPU, logging each instruction and register changes. The trace runs synchronously while the monitor is active. `trace 0` is rejected.

```
> trace 10
001000: move.l r1, #$2A          R1=$2A
001008: add.l r2, r1, #$10       R2=$3A
...
Trace complete: 10 instructions
```

If a breakpoint is hit during tracing, the trace stops early:

```
> trace 1000
...
Trace stopped at breakpoint $1010
Trace complete: 1000 instructions
```

The completion line reports the requested count, even if a breakpoint stopped the trace early.

`trace file <path>` / `trace file off` - File Output

Direct trace output to a file instead of the scrollback buffer. Use `trace file off` to resume scrollback output.

```
> trace file trace.log
Trace output to trace.log

> trace 10000

> trace file off
Trace file output stopped
```

Trace files are synced before they are closed. When trace output is redirected to a file, the final `Trace complete: N instructions` line still appears in the monitor scrollback.

`trace watch add <addr>` - Track Memory Writes

Add a memory address to the write-tracking list. Trace watches are byte watches: after each traced instruction, the monitor reads one byte at each tracked address and records a write if the byte differs from the previous sampled value.

```
> trace watch add $1000
Trace watch added at $1000
```

`trace watch del <addr>` - Stop Tracking

Remove an address from the write-tracking list.

```
> trace watch del $1000
Trace watch removed at $1000
```

`trace watch list` - List Tracked Addresses

```
> trace watch list
  $1000
  $2000
```

`trace history show <addr>` - Show Write History

Display recorded writes to a tracked address, including the PC that performed the write, and old/new values. Step numbers are relative to the trace run that recorded the write. The history is capped at 256 records per address; when the cap is exceeded, older records are dropped.

```
> trace history show $1000
$1000: 4 writes recorded
  Step #42   PC=$001234  $00 -> $FF
  Step #108  PC=$005678  $FF -> $42
  Step #203  PC=$001234  $42 -> $00
  Step #510  PC=$009ABC  $00 -> $01
```

`trace history clear <addr|*>` - Clear History

Clear write history for a specific address or all addresses.

```
> trace history clear $1000
> trace history clear *
```

`trace mmio <region> [count]` - Show Region Access Events

Show the most recent access-history events whose address range overlaps a named memory or MMIO region for the focussed CPU. The command uses the monitor region registry for the current CPU; unknown region names are rejected. If `count` is omitted, the monitor prints up to 16 matching events. Count parsing uses the same count syntax as `trace <count>`.

```
> trace mmio mmio
> trace mmio vram 32
```

The command requires the debug access service. If the service is unavailable, no CPU is focussed, the region is unknown, or the count is invalid, IEMon prints an error and does not change execution state.

I/O Register Viewer

`io [device]` - View I/O Registers

Display formatted I/O register values for a hardware device. Without arguments, lists available devices. Use `io all` to dump every device's registers at once.

```
> io
Available I/O devices:
  ahx
  antic
  arosdos
  audio
  boothostfs
  clipboard
  coproc
  cpu_wait
  exec
  fileio
  gtia
  hosthelper
  irqdiag
  media
  midilive
  midiplay
  mod
  paula
  pokey
  psg
  sid
  sid2
  sid3
  sn76489
  sfx
  sysinfo
  ted
  terminal
  ula
  vga
  video
  voodoo
  voodoo_depth
  wav

> io vga
--- VGA Registers ---
  MODE                ($F1000) = $00000013 [19] RW
  STATUS              ($F1004) = $00000000 [0] RO
  CTRL                ($F1008) = $00000001 [1] RW
...

> io all
(dumps all listed devices)
```

Register widths are per-register (1, 2, 4, or 8 bytes). Values are displayed in the appropriate width with both hex and decimal representations, followed by the access mode (`RO`, `WO`, or `RW`). The monitor and Lua `dbg.io()` use the same native-width MMIO read path for these registers, including when the focussed CPU is 6502, so word, long, and quadword

registers are not reconstructed through byte reads from the CPU memory view. Unknown devices print `Unknown device: <name>`.

The audio/player views mirror the MMIO layout. `psg`, `sid`, `ted`, and `pokey` are combined chip/player views, so their playback control registers are shown alongside chip registers. `ahx`, `midiplay`, `mod`, and `wav` are independent player/control views. `midiplay` is the file-backed SMF `.mid`/`.midi` and Doom `.mus` player backed by the `MIDI_PLAY_*` MMIO block.

`sfx` shows the trigger-channel sample MMIO block using the extended 32-channel window. Channels `CH0` through `CH31` each show `PTR`, `LEN`, `LOOP_PTR`, `LOOP_LEN`, `FREQ`, `VOL`, `PAN`, `FORMAT`, and `CTRL`; the legacy four-channel aliases address the same first four channels. Bridge and profile integration views such as `hosthelper`, `arosdos`, `paula`, `clipboard`, and `boothostfs` expose their register blocks; values depend on the active runtime/profile and may indicate disabled or idle state. `paula` is the Paula-style DMA shim, and `boothostfs` is Bootstrap HostFS.

`midilive` is the generic live-MIDI port, the CPU-agnostic MMIO stream that drives the shared synth from any core or from EhBASIC, separate from the file-backed `midiplay`. It exposes `LIVE_DATA` (`$F0BF4`, write a raw MIDI byte), `LIVE_STATUS` (`$F0BF5`, read bit 0 = live port active), and `LIVE_CTRL` (`$F0BF6`, write bit 0 = reset).

```
> io midilive
--- Live MIDI Port Registers ---
  LIVE_DATA            ($F0BF4) = $00       [0] WO
  LIVE_STATUS          ($F0BF5) = $01       [1] RO
  LIVE_CTRL            ($F0BF6) = $00       [0] WO
...

> io midiplay
--- MIDI/MUS Player Registers ---
  PLAY_PTR             ($F0BA0) = $00000000 [0] RW
...

> io mod
--- MOD Player Registers ---
  FILTER_MODEL         ($F0BD0) = $00000000 [0] RW
...

> io paula
--- Paula DMA Registers ---
  CH0_PTR              ($F2260) = $00000000 [0] RW
...
```

Hex Editor

`e <addr>` - Enter Hex Editor

Open an interactive hex editor at the specified address. The display switches to a hex grid showing up to 16 rows of 16 bytes (256 bytes total on the default-size overlay).

```
> e $1000
```

Hex Editor Controls:

Key	Action
Arrow keys	Move cursor
0-9, A-F	Edit current nibble

Key	Action
PgUp/PgDn	Scroll by 256 bytes
Enter	Commit changes to memory
Esc	Discard changes and return to command mode

Changed bytes are highlighted in green. The cursor position is shown with inverted colours. Edits are kept in a dirty map until Enter commits them; Esc discards the dirty bytes. If the focussed CPU adapter rejects an address while editing, that byte is left unchanged.

Scripting

`script <filename>` - Run Command Script

Execute monitor commands from a text file, one command per line. After leading and trailing whitespace is removed, lines starting with # are treated as comments and skipped. Inline comments are not stripped.

```
> script setup.mon
```

Example script file (`setup.mon`):

```
# Set up breakpoints for debugging
b $1000
b $2000 r1==$FF
ww $3000
trace watch add $3000
```

Scripts can nest up to 8 levels deep. Semicolon-separated commands on a script line are supported outside quotes. Monitor command scripts are more powerful than trusted `.iemonrc` files: they can run ordinary monitor commands and should be used only with files you intend to execute.

`macro <name> <commands>` - Define Macro

Define a named macro as a semicolon-separated list of commands. Invoke the macro by typing its name.

```
> macro inspect r ; d ; m sp 4
Macro 'inspect' defined (3 commands)

> inspect
(runs r, then d, then m sp 4)
```

Macros persist for the duration of the session. Macro names are case-insensitive (the name is lowercased on definition and lookup). Redefining an existing macro silently overwrites it; there is no `macro list` or `macro del` command. Macros share the script nesting counter, so recursive invocation aborts with `Macro recursion limit reached` once depth exceeds 8.

Audio Control

`fa` - Freeze Audio

Freeze audio playback. By default, audio continues playing while the monitor is open (it's output-only and doesn't affect memory state). Use this command to silence audio during debugging.

```
> fa
Audio frozen
```

`ta` - Thaw Audio

Resume audio playback.

```
> ta
Audio thawed
```

Help

`?` / `help` - Command Reference

Display the command reference. `help` lists every registered command and `help <command>` prints syntax plus worked examples from the same registry that drives the in-monitor help text.

Scripted equivalent: `dbg.help([name])`.

```
> help pg
pg - Add, list, or clear page-access guards
Syntax:
  pg add <start> <end> <rwx> [cpu=all|current]
  pg list
  pg clear
Examples:
  pg add $4000 $4FFF rw cpu=current
  pg add code code+255 x
  pg list
```

Every command in the monitor help registry is expected to have at least one example; this is covered by the IEMon UX tests. Dispatch-level aliases that are documented separately, such as `wr` and `wrw`, are not separate help-registry entries.

Command History, Aliases, Layouts, and Reports

Command history is kept in memory for the overlay and persisted to `~/.iemon/history` in interactive builds. Duplicate adjacent commands are collapsed, and the on-disk history is capped so long debugging sessions do not grow it without bound. In tests, persistence is disabled unless `IEMON_HOME` is set explicitly.

Pressing Enter on an empty command line repeats the last repeatable command (`s`, `d`, or `m`). This is intended for short step/disassemble/memory workflows where repeatedly typing the same command is noise.

Aliases are session-local command shorthands:

```
> alias ni s
Alias ni = s

> alias regs r
Alias regs = r

> alias
ni = s
regs = r
```

Aliases cannot replace built-in command names.

Named layouts render common inspection views without changing emulator state:

Command	View
`layout cpu`	Registers plus nearby disassembly
`layout trace`	Instruction trace ring tail plus recent access events
`layout debug`	Registers, disassembly, stack, and page guards
`layout list`	Available built-in layouts
`layout save <name>`	Saves a session alias for the debug layout

The `bug [trace-count]` command prints a copyable report bundle containing the focussed CPU, registers, disassembly, stack, memory regions, page guards, access events, trace-ring entries, and loaded symbols. It is deliberately plain text so it can be pasted into an issue without extra formatting work.

Scripted equivalents: `dbg.layout(name)` and `dbg.bug_report([trace_count])`.

Keyboard Shortcuts

Key	Action
Enter	Submit command
Esc	Exit monitor in command mode; discard changes and return to command mode in the hex editor
Up/Down	Navigate command history
Left/Right	Move cursor within input line
Home / End	Jump to start/end of input line
Delete	Delete character at cursor
Backspace	Delete character before cursor
Ctrl+A / Ctrl+E	Jump to start/end of input line
Ctrl+K	Kill from cursor to end of line
Ctrl+U	Kill from start of line to cursor
Ctrl+Left/Right	Jump by word
Ctrl+Shift+V	Paste from clipboard
Shift+Left/Right	Extend selection by one character
Shift+Up/Down	Extend selection by one line
Shift+Home/End	Extend selection to start/end of line
Ctrl+Shift+C	Copy selected text to clipboard
Ctrl+Shift+X	Cut selected text (input line only)
Middle mouse button	Paste selection (or clipboard if nothing selected)
PgUp/PgDn	Scroll output buffer
Mouse wheel	Scroll output buffer

Key	Action
F9	Toggle monitor on/off
F10	Hard reset (works while monitor is active)

Cursor movement, delete, and backspace keys repeat automatically when held.

CPU-Specific Notes

IE64 (64-bit RISC)

- 32 general-purpose 64-bit registers: R0-R31
- R0 is always zero, R31 is the stack pointer (SP)
- Fixed 8-byte instruction encoding
- Register display: 16-digit hex (`$0000000000001000`)

IE32 (32-bit RISC)

- 16 general-purpose 32-bit registers: A, X, Y, Z, B, C, D, E, F, G, H, S, T, U, V, W (note the gap: I through R are not register names)
- Plus PC and SP, for 18 registers total in the snapshot
- Fixed 8-byte instruction encoding
- Register display: 8-digit hex (`$00001000`)

M68K (Motorola 68020)

- Data registers D0-D7, address registers A0-A7
- A7 is the stack pointer, A6 is typically the frame pointer
- SR (status register), USP (user stack pointer), SSP (supervisor stack pointer)
- Variable-length instructions (2-22 bytes; 68020 indirect/scaled-index modes can extend a 2-byte opcode by up to 20 bytes of extension words)
- Big-endian byte order

Z80

- Main registers: A, F, B, C, D, E, H, L
- Shadow registers: A', F', B', C', D', E', H', L'
- Index registers: IX, IY
- Other: SP, PC, I (interrupt vector), R (refresh), IM (interrupt mode)
- Register display: 4-digit hex (`$0040`)

6502 (MOS Technology)

- A (accumulator), X, Y (index registers)
- SP (stack pointer, 8-bit), PC (program counter, 16-bit)
- SR (status register with N/V/-/B/D/I/Z/C flags)
- All instructions are 1-3 bytes

X86 (Intel 32-bit)

- General: EAX, EBX, ECX, EDX
- Index: ESI, EDI
- Pointer: ESP, EBP
- EIP (instruction pointer), EFLAGS
- Segment registers (16-bit): CS, DS, ES, SS, FS, GS
- Variable-length instructions (1-15 bytes)

Multi-CPU Debugging Workflows

Debugging a Coprocessor

1. Press F9 to enter the monitor
2. `cpu` to list all CPUs - coprocessors appear automatically
3. `cpu 1` to focus on the coprocessor
4. `r` to inspect registers, `d` to disassemble
5. `s` to single-step the coprocessor
6. `cpu 0` to switch back to the primary CPU
7. `x` to resume the CPUs that were running when the monitor was entered

Stepping One CPU While Others Run

1. Press F9 (running CPUs are frozen)
2. `thaw 1` - let the coprocessor run freely
3. `s` - step the primary CPU while coprocessor executes
4. `freeze *` - re-freeze everything to inspect shared state
5. `m $3000 4` - examine shared memory

Setting a Breakpoint and Continuing

1. `b $2000` - set breakpoint at address $2000
2. `x` - exit monitor and resume
3. When execution reaches $2000, the monitor activates automatically
4. `r` - inspect the state at the breakpoint
5. `bc $2000` - clear the breakpoint
6. `x` - resume the CPUs that were running when the monitor was entered

Using Conditional Breakpoints

1. `b $1000 r1==$FF` - break only when R1 is 0xFF
2. `x` - resume execution
3. The breakpoint is checked each time PC reaches $1000, but only fires when R1 equals 0xFF

Tracing a Memory Write

1. `trace watch add $3000` - track writes to $3000
2. `trace 1000` - trace 1000 instructions (decimal)
3. `trace history show $3000` - see which instructions wrote to $3000 and what values they wrote

Saving and Restoring State

1. `ss checkpoint.iem` - save current state
2. (debug, modify registers, step through code)
3. `sl checkpoint.iem` - restore to the saved state

Using Macros for Repetitive Tasks

1. `macro dump r ; d ; m sp 4 ; bt` - define a macro that shows registers, disassembly, stack memory, and backtrace
2. `s` - step an instruction
3. `dump` - run the macro to inspect everything at once

Display

The monitor overlay is a character grid sized to the current native video mode (`screenWidth / 8` columns x `screenHeight / 16` rows, for example 120x33 at the 960x540 default), rendered with the Amiga Topaz 8 bitmap font. Desktop presentation scales that native surface into the default 1920x1080 fullscreen output. Colours follow classic monitor conventions:

- **White**: Default text
- **Cyan**: Headers, labels, informational messages
- **Yellow**: Current PC line in disassembly
- **Red**: Breakpoint markers, error messages
- **Green**: Changed register values, modified bytes in hex editor
- **Magenta**: Backward branch / loop markers
- **Dim blue**: Inactive/separator text

IE64 Fault Interception

Fault Interception

`fault` lets IEMon break when a CPU fault or exception is detected. It is off by default so the normal CPU exception path continues unless interception is enabled.

```
> fault list
Fault interception: off

> fault break ie64.priv
Fault break enabled: ie64.priv

> fault on
Fault interception enabled for all supported faults

> fault off
Fault interception disabled
```

Supported fault kinds:

CPU	Kinds
IE64	`ie64.not-present`, `ie64.read-denied`, `ie64.write-denied`, `ie64.exec-denied`, `ie64.user-supervisor`, `ie64.priv`, `ie64.syscall`, `ie64.misaligned`, `ie64.illegal`, `ie64.skef`, `ie64.skac`
IE32	`ie32.invalid-opcode`
M68K	`m68k.bus-error`, `m68k.address-error`, `m68k.illegal`, `m68k.divide-zero`, `m68k.privilege`, `m68k.line-a`, `m68k.line-f`, `m68k.format-error`, `m68k.trapv`
Z80	`z80.rst38`, `z80.nmi`
6502	`6502.brk`
X86	`x86.ud`

The cause-code table below is reproduced here for convenience when interpreting IE64 fault interception and stopped-CPU state. For the processor reference context, see `IE64_ISA.md` section 11.8.

Cause	Label	Trigger
0	`page-not-present`	Absent PTE mapping or unavailable physical/atomic backing
1	`read-denied`	PTE `R==0` on load
2	`write-denied`	PTE `W==0` on store
3	`exec-denied`	PTE `X==0` on instruction fetch
4	`user-supervisor`	User mode access to `PTE_U==0` page
5	`priv`	User-mode execution of a privileged instruction
6	`syscall`	`SYSCALL` instruction
7	`misaligned`	Atomic RMW with misaligned address
8	`timer`	Timer interrupt (via INTR_VEC)
9	`skef`	Supervisor instruction fetch from user page (`MMU_CTRL.SKEF`)
10	`skac`	Supervisor data access to user page with `MMU_CTRL.SKAC` set and `MMU_CTRL.SUA` clear
11	`illegal`	Opcode-level invariant or illegal-instruction trap, currently including `MTCR` to read-only `CR_RAM_SIZE_BYTES`

The CPU raises `FAULT_SKEF` (9) and `FAULT_SKAC` (10) with the correct numeric cause value. Use the numeric cause when triaging supervisor-guard faults.

`SKEF` and `SKAC` usually indicate a supervisor-mode memory access bug: either a stray supervisor fetch into a user page or a missing `SUAEN` / `SUADIS` bracket around supervisor access to a user page.

The monitor's IE64 disassembler recognises the `suaen` and `suadis` mnemonics, so disassembly listings show the helper brackets at their real source locations rather than as raw `dc.b $F3` or `dc.b $F4`.

Common Pitfalls

- **Bare `d` counts are hexadecimal.** `d $1000 10` shows 0x10 = 16 lines, not 10. Use `d $1000 #10` for decimal 10. Counts for `s`, `m`, `trace`, and `bt` are decimal.

- **`#N` does not work for most count arguments.** It is honoured for address/value parsing and for the `d` line count because `d` uses the address parser for that argument. For other counts, use bare decimal, `$hex`, or `0xhex`. For example, `s #10` silently steps one instruction because the invalid count is ignored.
- **Legacy `ww` is a one-byte write watchpoint.** Use `bpm*` commands for read, write, read/write, and wider overlapping access watchpoints.
- **Access-backed watchpoints require instrumentation.** In normal supported builds the CPU access hooks drive read/write/fetch watchpoints. When access instrumentation is not enabled, access-backed commands fail closed instead of advertising partial behaviour.
- **`ss`/`sl` are focussed-CPU only.** Other CPUs, video/audio/timer device state, and coprocessor state are not in the snapshot. `sl` will refuse a snapshot whose CPU type differs from the focussed CPU.
- **`bs` (backstep) is focussed-CPU only.** It restores that adapter's registers and memory view from the per-CPU ring (max 32 entries). It does not roll back device state or other CPUs.
- **M68K backtrace prefers A6 frame links.** Code that does not use LINK/UNLK falls back to an A7 stack scan, so symbol coverage strongly affects how much noise is filtered.
- **6502 backtrace is heuristic.** Each frame is tagged `(low confidence)`; the walker scans page 1 upward from SP+1 and assumes JSR-pushed return addresses.
- **`bc *` / `wc *` clear only the focussed CPU.** Switch CPUs and repeat to clear globally; `bl` lists across all CPUs but `bc *` does not.
- **Run-until can leave a temp breakpoint if never hit.** If `u` created a new temporary breakpoint and the PC never reaches the target before you re-enter the monitor for another reason, clear it explicitly with `bc <addr>`.
- **`g <addr>` silently ignores parse errors** and resumes from the current PC. Use `r pc <addr>` first if you want a hard error on bad input. If the address parses but lies outside the focussed CPU's address space, `g` prints a red `ValidateAddress` error and stays in the monitor.
- **`trace` breakpoint checks happen after each step.** Normal run breakpoints stop before executing the instruction at the breakpoint PC. `trace` steps first, then checks whether the new PC is a breakpoint.
- **`trace history` records are capped at 256 per address.** Older writes are dropped FIFO; only the latest 256 records for that address are retained.
- **`h` and `c` cap displayed hits at 256** before printing `... (truncated)`. Scanning stops once that display limit is reached.
- **`f` (fill) refuses ranges over 1 MiB** with `Invalid range`. Use repeated `f` calls for larger spans.
- **`r <name> <value>` does not accept expressions.** The value goes through `ParseAddress`, so `$N`, `0xN`, bare hex, and `#decimal` work, but `pc+8` does not. Use `r` only to set a literal value.
- **Not every displayed register is writable.** For example, IE64 `R0` is hardwired to zero, and the Z80 shadow registers are displayed but are not accepted by `r <name> <value>` or expression evaluation.
- **Invalid optional arguments are often ignored.** `d bad`, `d $1000 bad`, `m $1000 bad`, `s bad`, and `bt bad` keep their defaults rather than printing parse errors.
- **Raw memory tools trust the CPU adapter.** `save`, `c`, and `t` do not perform a separate address validation pass over the full range. For out-of-map ranges, results follow the focussed adapter's read/write behaviour.