

## Chapter 33 - IE Mon

---

IE Mon is the interactive monitor built into Intuition Engine. It runs in the same terminal as BASIC and gives you direct access to the bus, the active CPU's registers, breakpoints, watchpoints, a disassembler, a byte memory editor, a reverse-execution timeline, save and restore of memory ranges, and a small handful of more specialised tools.

You enter the monitor from BASIC by typing `MON` at the prompt and leave it by typing `x`. Inside the monitor, each line is a command followed by space-separated arguments. Numeric arguments default to hexadecimal; prefix them with `#` for decimal.

### 33.1 General conventions

Convention	Example	Meaning
Address	1000	Hexadecimal \$1000
Address	#4096	Decimal
Address pair	1000 10FF	Inclusive range
Register	r0, pc, a	Active CPU's register
Hex byte literal	7E	Hex byte for write/fill

The monitor's prompt is the active CPU's short name followed by a greater-than sign: (ie64)>, (6502)>, (z80)>, etc.

Many address arguments accept a small expression form. The terms may be numbers, registers, or loaded symbols, joined with `+` or `-`. This applies to `d`, `list`, `m`, `b`, `g`, `u`, `ww`, `wc`, `addr`, `who`, `trace watch`, `trace history`, `e`, the start and end addresses in `save`, and the destination address in `load`. It also applies to `sym add`, `sym resolve`, the optional base in `sym loadlbl`, and the start and end addresses in `pg addr`. The byte-range commands `w`, `f`, `h`, `c`, `t`, and `bc` take literal addresses. IE64 assemble mode A also takes a checked literal physical address.

### 33.2 Execution control

Command	Argument(s)	Effect
<code>g</code>	[addr]	Resume execution from current PC or <code>addr</code>
<code>s</code>	[count]	Step: execute one instruction (or <code>count</code> )
<code>u</code>	<code>addr</code>	Run until: resume, stop one-shot at <code>addr</code>
<code>x</code>		Exit the monitor (return to BASIC)

`s` always re-prints the registers after the step. When stepping, processor `WAIT` instructions count as one instruction and do not sleep; `g` resumes normal timing. `g` returns to the monitor on the next breakpoint, watchpoint, or `Ctrl-C`.

## 33.3 Registers and disassembly

Command	Argument(s)	Effect
r	[reg] [value]	Show all registers, or set one register
d	[addr] [count]	Disassemble at <code>addr</code>
list	[addr] [count]	List annotated lines when line data is loaded
cpu	[name]	Show the active CPU or switch to <code>name</code>

The argument to `cpu` is `ie64`, `ie32`, `6502`, `z80`, `m68k`, or `x86`.

## 33.4 Memory inspection and editing

Command	Argument(s)	Effect
m	addr [count]	Memory dump in hex + ASCII
w	addr byte [byte...]	Write bytes at <code>addr</code>
A	addr	IE64 assemble one instruction at <code>addr</code>
e	addr	Enter interactive hex editor mode
f	start end byte	Fill a range with one byte
h	start end pattern...	Hunt: search a range for a byte pattern
c	start end dest	Compare a range against <code>dest</code>
t	start end dest	Transfer a range to <code>dest</code>

`w` is a one-line byte writer. Use `e addr` when you want interactive byte editing. Byte arguments should be `00` to `FF`; larger values are stored as their low byte.

`A addr` enters IE64-only one-instruction assemble mode at a checked unsigned physical address. The prompt becomes `asm $0000000000001000>` and each non-empty line must assemble to exactly one 8-byte IE64 instruction. On success the monitor writes physical RAM only, prints the address, bytes, and disassembly, advances by eight bytes, and makes the new instruction visible to IE64 execution. Errors leave memory and the current assemble address unchanged. Press Enter on an empty line to exit.

Monitor assemble mode rejects source-file features: labels, directives, `include`, `incbin`, output formats, files, and multi-instruction pseudo-ops such as `li`. It is a monitor entry tool for short IE64 code, patches, and examples, not a stored source-file language. Scripts, macros, `.iemonrc` files, and IEScript raw monitor wrappers cannot enter or feed assemble mode.

### 33.4.1 Byte-entry audio workflow

This is the standard machine-language programming loop in IE Mon:

```

(6502)> w 1000 A9 01 8D 00 F8 A9 00 8D 08 D2 A9 79 8D 00 D2 A9
(6502)> w 1010 AF 8D 01 D2 4C 14 10
(6502)> d 1000 9
 1000: A9 01          LDA #$01
 1002: 8D 00 F8      STA $F800
 1005: A9 00          LDA #$00
 1007: 8D 08 D2      STA $D208
 100A: A9 79          LDA #$79
 100C: 8D 00 D2      STA $D200
 100F: A9 AF          LDA #$AF
 1011: 8D 01 D2      STA $D201
T 1014: 4C 14 10     JMP $1014
(6502)> r pc 1000
(6502)> b 1014
(6502)> g
(6502)> m D200 2
D200: 79 AF 00 00 00 00 00 00 00 00 C0 00 00 00 00 00  y.....
D210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
(6502)> bc 1014

```

The leading prefix in each disassembly line tells you what the monitor thinks about that address: two spaces for an ordinary instruction, T for a branch target that another line in the window references (the self-loop at \$1014 here), > for the current PC, and \* for an active breakpoint. The d count argument is parsed as hexadecimal by default (so d 1000 9 gives nine lines and d 1000 10 would give sixteen); prefix the count with # for decimal. The m count argument is decimal by default and counts *rows of sixteen bytes*, not bytes. m D200 2 shows two sixteen-byte rows starting at \$D200, and the values written by the program appear in the first few columns of the first row.

The w line enters bytes. The d line proves those bytes decode as the intended instructions. The breakpoint stops execution before the final self-loop, and the memory dump proves the POKEY frequency and control bytes were written. You should hear the tone while the audio engine is enabled. If the disassembly does not match the chapter transcript, fix the byte listing before running it.

### 33.4.2 IE64 assemble workflow

For IE64 only, A addr lets you type one mnemonic instruction at a time. The monitor writes exactly one 8-byte instruction, prints the emitted bytes, then moves the assemble prompt forward by 8 bytes. That makes small IE64 programs easier to read without leaving the monitor.

This example writes one 32-bit value into RAM at \$2000, then stops at a self-loop:

```

(ie64)> A 1000
IE64 assemble at $000000000001000; empty line exits
asm $000000000001000> move.q r2,$2000
$000000000001000: 01 17 00 00 00 20 00 00  move.q r2, #$2000
asm $000000000001008> move.q r1,$12345678
$000000000001008: 01 0F 00 00 78 56 34 12  move.q r1, #$12345678
asm $000000000001010> store.l r1,(r2)
$000000000001010: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
asm $000000000001018> bra $1018
$000000000001018: 40 06 00 00 00 00 00 00  bra $001018
asm $000000000001020>
Exited IE64 assemble mode
(ie64)> d 1000 #4
 00001000: 01 17 00 00 00 20 00 00  move.q r2, #$2000
 00001008: 01 0F 00 00 78 56 34 12  move.q r1, #$12345678
 00001010: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
T 00001018: 40 06 00 00 00 00 00 00  bra $001018
(ie64)> r pc 1000
(ie64)> b 1018
(ie64)> g
(ie64)> m 2000 1
000000000002000: 78 56 34 12 00 00 00 00 00 00 00 00 00 00 00 00  xV4.....
(ie64)> bc 1018

```

The A lines are not separate source code. They are monitor commands. The bytes printed after each line are the same bytes we would have written. The d command is still the proof step, and the memory dump shows little-endian storage of \$12345678.

### 33.4.3 Byte-entry graphics workflow

The same loop works for visible hardware. This 6502 example uses the ULA card: it sets the border, enables the ULA source, writes an 8-byte bitmap motif through the ULA latch/data port, writes one attribute byte, and then loops. It is deliberately small, but it does more than poke a colour register: it teaches the latched VRAM entry path used by larger ULA programs.

```

(6502)> w 1100 A9 05 8D 00 D8 A9 05 8D 04 D8 A9 00 8D 0C D8 A9
(6502)> w 1110 00 8D 10 D8 A9 FF 8D 14 D8 A9 81 8D 14 D8 A9 BD
(6502)> w 1120 8D 14 D8 A9 A5 8D 14 D8 A9 A5 8D 14 D8 A9 BD 8D
(6502)> w 1130 14 D8 A9 81 8D 14 D8 A9 FF 8D 14 D8 A9 00 8D 0C
(6502)> w 1140 D8 A9 18 8D 10 D8 A9 46 8D 14 D8 4C 4B 11
(6502)> d 1100 #31
 1100: A9 05          LDA #$05
 1102: 8D 00 D8      STA $D800
 1105: A9 05          LDA #$05
 1107: 8D 04 D8      STA $D804
 110A: A9 00          LDA #$00
 110C: 8D 0C D8      STA $D80C
 110F: A9 00          LDA #$00
 1111: 8D 10 D8      STA $D810
 1114: A9 FF          LDA #$FF
 1116: 8D 14 D8      STA $D814
 1119: A9 81          LDA #$81
 111B: 8D 14 D8      STA $D814
 111E: A9 BD          LDA #$BD
 1120: 8D 14 D8      STA $D814
 1123: A9 A5          LDA #$A5
 1125: 8D 14 D8      STA $D814
 1128: A9 A5          LDA #$A5
 112A: 8D 14 D8      STA $D814
 112D: A9 BD          LDA #$BD
 112F: 8D 14 D8      STA $D814
 1132: A9 81          LDA #$81
 1134: 8D 14 D8      STA $D814
 1137: A9 FF          LDA #$FF
 1139: 8D 14 D8      STA $D814
 113C: A9 00          LDA #$00
 113E: 8D 0C D8      STA $D80C
 1141: A9 18          LDA #$18
 1143: 8D 10 D8      STA $D810
 1146: A9 46          LDA #$46
 1148: 8D 14 D8      STA $D814
T 114B: 4C 4B 11     JMP $114B
(6502)> r pc 1100
(6502)> b 114B
(6502)> g
(6502)> m D800 1
D800: 05 00 00 00 05 00 00 00 01 00 00 00 02 00 00 00 .....
(6502)> bc 114B

```

The writes to \$D80C and \$D810 set the low and high halves of the ULA VRAM address latch. Each store to \$D814 writes one byte at the latched address and advances the latch. The first eight data bytes form the bitmap motif. The later latch value \$1800 selects the attribute area, and \$46 gives the cell yellow ink on black paper. The d count of #31 is decimal (the # prefix forces decimal. Without it the count is parsed as hexadecimal, which is the default for d); the single m D800 1 shows the sixteen-byte row that starts at \$D800 so the writes to \$D800 (05) and \$D804 (05) appear in the first row. The \$D814 register is a write-only auto-incrementing latch port; reading it does not step through the bitmap data the program wrote, so the verification is the visible stripe on the ULA display, not a follow-on m D814 read. Try changing the first \$A5 byte to \$99; the middle of the motif changes without moving the attribute cell.

## 33.5 Breakpoints

Command	Argument(s)	Effect
b	addr	Set a code breakpoint at <code>addr</code>
bc	[ <code>addr</code>   <code>id</code> ]	Clear a breakpoint by address or by ID
bl		List all breakpoints
bfirst		Break on the first hit only (set policy flag)

Breakpoints are sticky: setting one a second time at the same address increments a hit-counter rather than creating a duplicate.

## 33.6 Watchpoints (memory)

The monitor distinguishes between **byte**, **word**, **dword**, and **qword** watchpoints, and between **read**, **write**, and **read/write** flavours. The mnemonic encoding is:

```
bpm <size> <mode>      where size = b/w/d/q and mode = r/w/(both)
```

Command	Effect
ww addr	Word watchpoint, read/write (legacy alias)
wr addr	Word watchpoint, read only
wrw addr	Word watchpoint, read only (verbose alias)
bpmbr addr	Byte read
bpmbw addr	Byte write
bpm b addr	Byte read/write
bpmwr addr	Word read
bpmww addr	Word write
bpmw addr	Word read/write
bpm dr addr	Dword read
bpm dw addr	Dword write
bpm d addr	Dword read/write
bpmqr addr	Qword read
bpmqw addr	Qword write
bpmq addr	Qword read/write
wc [ <code>addr</code>   <code>id</code> ]	Clear a watchpoint
wl	List all watchpoints

These are memory watchpoints: they do not write anything. The similar-looking ww/wr are the abbreviations sometimes confused with byte writers - they are not.

## 33.7 Reverse execution

The monitor keeps two kinds of history. `bs` and `rs` use a CPU-local step snapshot for the focussed CPU only. `rg`, `rt`, `tl`, and `history` use the retained whole-machine reverse-history timeline, which includes registered CPUs, bus RAM, backed RAM, and versioned device snapshots.

Command	Argument(s)	Effect
<code>bs</code>	<code>[count]</code>	Back-step: undo one (or <code>count</code> ) instructions
<code>rs</code>	<code>[count]</code>	Same as <code>bs</code>
<code>rg</code>		Reverse-continue: rewind until the previous breakpoint or watchpoint
<code>rt</code>	<code>addr</code>	Reverse-run-until: rewind until PC was <code>addr</code>
<code>tl</code>		Timeline: dump the recent history
<code>history</code>		Same as <code>tl</code>

Be careful: `rg` is **reverse-continue**, not "register inspect".

## 33.8 Tracing

Command	Argument(s)	Effect
<code>trace</code>	<code>on off name</code>	Enable or disable instruction trace
<code>trace mmio</code>	<code>region [count]</code>	Show recent access events in a named MMIO or memory region
<code>tracing</code>	<code>[size]</code>	Enable a ring-buffer trace of recent instructions
<code>show</code>		Dump the ring trace

`t` race is the named trace command. It is distinct from `t` (which is **transfer memory**, not trace).

## 33.9 CPU freeze and thaw

These commands stop and resume execution of a specific CPU without exiting the monitor.

Command	Argument(s)	Effect
<code>freeze</code>	<code>cpuName   *</code>	Freeze one CPU, or <code>*</code> for all
<code>thaw</code>	<code>cpuName   *</code>	Thaw one CPU, or <code>*</code> for all
<code>fa</code>		Freeze audio (mixer + all engines)
<code>ta</code>		Thaw audio

`fa` and `ta` are **audio-only** controls, not freeze-all aliases. To freeze every CPU at once, use `freeze *`.

## 33.10 Save and restore

Command	Argument(s)	Effect
<code>save</code>	<code>start end name</code>	Save a memory range

Command	Argument(s)	Effect
load	name addr	Load a memory range at <code>addr</code>
ss	name	Save CPU-local state
sł	name	Load CPU-local state

save and load move memory ranges. `ss` and `sł` save and restore the focussed CPU adapter's registers plus its fixed CPU-local memory snapshot span. They do not save other CPUs, audio envelopes, video state, timers, DMA engines, coprocessor state, or monitor reverse-history state. `sł` refuses a snapshot saved for another CPU type. For whole-machine reverse debugging, use `rg`, `rt`, `tl`, and `history`; that history is a retained timeline, not a permanent file format.

## 33.11 Symbols, addresses, maps

Command	Argument(s)	Effect
sym	[name   addr]	Look up a symbol or address
map		Show the active loaded symbol map
addr	addr	Resolve <code>addr</code> to a symbol if any

`sym add name addr [kind]` records a symbol for the active CPU. `sym resolve addr` searches by address. `sym loadlbl name [base]` loads a label file and optionally relocates it by `base`. The address operands in these forms accept the expression syntax from section 33.1; symbol names, kinds, and filenames are plain text arguments.

## 33.12 Bus, MMIO, page-level access

Command	Argument(s)	Effect
io	[device   all]	List I/O views or show register values
pg	add start end mode	Page guard: trap on access to a range
pg	list   clear	List or clear page guards
accesslog	on off show [count]	Control per-page access logging
who	read wrote fetched addr	Report the last matching access
trace mmio	region [count]	Show recent access events in a named region

`io` is an I/O register viewer. With no argument it lists the named views known to the monitor. `io all` prints every listed view. `io name` prints the registers in one view, using the register width from the monitor's I/O table. A byte register is shown as two hex digits, a word as four, and a long as eight, followed by decimal value and access mode.

The monitor reads these registers through the native bus width for the register, not through the active CPU's ordinary byte memory view. That matters when the focussed CPU is a 6502 or another narrow client: a long player pointer is still read as one long register on the shared bus.

Useful view groups are:

Group	Views
Core machine	video, terminal, audio, fileio, media, exec, coproc, sysinfo, irqdiag
Video cards	vga, ted, antic, gtia, ula, voodoo, voodoo_depth

Group	Views
Sound chips and players	ahx, midiplay, midilive, mod, wav, sn76489, psg, pokey, sid, sid2, sid3, sfx, paula
Bridge/profile inspection	arosdos, clipboard, and other service bridge views shown by io

The player views mirror their MMIO control blocks. `midiplay` shows the MIDI/MUS file-player registers, including `TEMPO_BPM`. `midilive` shows the byte-wide live MIDI stream port. `mod` and `wav` show the MOD and WAV player blocks. `sfx` shows the 32 extended trigger-channel sample registers. `psg`, `sid`, `ted`, and `pokey` are combined chip/player views, so their playback registers appear beside their chip registers.

Bridge/profile views are inspection aids for machine services that may be idle or unavailable in a particular running profile. Use the BASIC keywords and main chapters for ordinary file, HOST, and coprocessor work; use `io` when you need to see the register state.

```
(ie64)> io midiplay
--- MIDI/MUS Player Registers ---
PLAY_PTR          ($F0BA0) = $00000000 [0] RW
PLAY_LEN          ($F0BA4) = $00000000 [0] RW
PLAY_CTRL         ($F0BA8) = $00000000 [0] RW
PLAY_STATUS       ($F0BAC) = $00000000 [0] RO
POSITION          ($F0BB0) = $00000000 [0] RO
VOLUME            ($F0BB4) = $000000FF [255] RW
TEMPO_BPM         ($F0BB8) = $00000000 [0] RO
```

The exact values depend on what the player is doing. The important point is that the monitor shows each register at its natural width, so this view is a safe way to inspect mixed byte, word, and long MMIO blocks.

```
(ie64)> io midilive
--- Live MIDI Port Registers ---
LIVE_DATA         ($F0BF4) = $00 [0] WO
LIVE_STATUS       ($F0BF5) = $00 [0] RO
LIVE_CTRL         ($F0BF6) = $00 [0] WO
```

The live view is useful after a BASIC MIDI NOTE or a raw byte-stream test. `LIVE_STATUS` bit 0 is the proof that the live port is active. The two write-only registers normally read as zero.

## 33.13 Backtrace

Command	Argument(s)	Effect
bt		Stack backtrace from the active CPU

## 33.14 Command Results and Limits

Commands that cannot parse an address, register, CPU name, or byte value print an error line and leave the monitor active. `d`, `m`, `r`, `io`, `bt`, `tl`, `wl`, and `bl` inspect state. `w`, `f`, `t`, `load`, `sl`, register writes through `r`, and execution commands can change state.

`d` disassembles bytes; it never changes the program counter. `g` leaves the monitor until a breakpoint, watchpoint, manual break-in, or CPU stop brings control back. The reverse timeline is a recent-history tool, not permanent storage; use `ss` when you need a reloadable state.

## 33.15 Quick Reference Card

---

The single-letter and short commands you will use most:

r	show registers	bt	backtrace
d [a]	disassemble	s [n]	step
m [a]	memory dump	g [a]	go
w a b...	write bytes	u a	run until
A a	IE64 assemble	e a	hex editor
b a	breakpoint set	bs [n]	back-step
bc [a id]	breakpoint clear	rg	reverse-continue
bl	breakpoint list	rt a	reverse-run-until
ww/wr a	word watchpoint	tl	timeline
wc [a id]	watchpoint clear	x	exit
wl	watchpoint list	cpu n	switch CPU
freeze *	freeze all CPUs	thaw *	thaw all
fa / ta	freeze / thaw audio	save a b n	save range
ss / sl n	save / load CPU state	load n a	load range at a

## 33.16 What Comes Next

---

Chapter 34 covers IE Script, a command language that drives the monitor from stored command text rather than from a terminal. IE Script is what you use to automate session setup, breakpoint actions, data dumps after a fault, and similar batch debugging tasks.