

Chapter 34 - IE Script

IE Script is the batch-control partner to IE Mon. IE Mon is typed one command at a time; IE Script stores a sequence of commands and runs them as one job. Use it for repeatable setup, frame waits, VideoChip checks, breakpoint sessions, and fault capture.

IE Script is not the normal way to write CPU programs. BASIC and IE Mon remain the native programming route. IE Script automates the machine while a program is running.

34.1 Running a Script

A stored script uses the .IES suffix. From BASIC, run it with:

```
RUN "FIRST.IES"
```

From IE Mon, run the same stored script with:

```
(ie64)> script FIRST.IES
```

When a script finishes, BASIC or IE Mon regains control. If it raises an error, the message is printed and control returns to the surface that launched it.

34.2 Language Shape

An IE Script is a sequence of statements. It supports integers, booleans, strings, arithmetic, comparison, tables, functions, if/then/elseif/else/end, while, and for.

Comments begin with --:

```
-- Wait for two video frames, then print a line.  
sys.wait_frames(2)  
sys.print("ready")
```

Function names are lower case. Module names are lower case. Strings may use single or double quotes.

34.3 System Module

sys handles time, text output, script exit, and storage helpers.

Function	Purpose
sys.wait_frames(n)	Yield until n video frames have passed.
sys.wait_ms(ms)	Yield until ms milliseconds have passed.
sys.print(text)	Print text to the terminal.
sys.log(text)	Write a diagnostic line.

Function	Purpose
<code>sys.time_ms()</code>	Current monotonic time in milliseconds.
<code>sys.frame_count()</code>	Number of completed frames.
<code>sys.frame_time()</code>	Most recent frame time.
<code>sys.fps()</code>	Current frame-rate estimate.
<code>sys.perf_report()</code>	Return the subsystem performance report string.
<code>sys.perf_reset()</code>	Clear subsystem performance counters.
<code>sys.quit()</code>	Stop the script and return to the caller.
<code>sys.exit(code)</code>	Stop Intuition Engine with status <code>code</code> .
<code>sys.mkdir(name)</code>	Create a directory in approved script storage.
<code>sys.read_file(name)</code>	Return stored bytes as a string.
<code>sys.write_file(name, data)</code>	Write bytes to approved script storage.
<code>sys.copy_file(from, to)</code>	Copy stored bytes.
<code>sys.capture_output(name)</code>	Start terminal output capture.
<code>sys.capture_output_off()</code>	Stop terminal output capture.

Long loops should call `sys.wait_frames` or `sys.wait_ms`; this keeps cancellation responsive.

`sys.perf_reset()` and `sys.perf_report()` are for repeatable measurement scripts. The report is empty when performance accounting is off, or when no instrumented subsystem path ran during the measured span. When it is non-empty, each line gives a bucket name, total time, operation count, and average time per operation. The current subsystem buckets include video frame work, audio pulls, slow bus Read32 and Write32, and Voodoo swap stages.

34.4 CPU Module

`cpu` controls the selected CPU:

Function	Purpose
<code>cpu.load(name)</code>	Load and start a stored CPU program.
<code>cpu.load_stopped(name)</code>	Load a stored CPU program but leave it stopped.
<code>cpu.reset()</code>	Reset the selected CPU.
<code>cpu.freeze()</code>	Stop the selected CPU for safe raw RAM access.
<code>cpu.resume()</code>	Resume after <code>cpu.freeze()</code> .
<code>cpu.start()</code>	Start execution.
<code>cpu.stop()</code>	Stop execution.
<code>cpu.is_running()</code>	Return true if the selected CPU is running.
<code>cpu.mode()</code>	Return the selected CPU name.
<code>cpu.execution_mode()</code>	Return the current execution mode name.

Raw RAM access through `mem` requires the CPU to be frozen. MMIO access is allowed while the CPU is running.

34.5 Memory Module

mem reads and writes the bus:

Function	Purpose
mem.read8(addr)	Read one byte.
mem.read16(addr)	Read one little-endian word.
mem.read32(addr)	Read one little-endian long.
mem.write8(addr, value)	Write one byte.
mem.write16(addr, value)	Write one word.
mem.write32(addr, value)	Write one long.
mem.read_block(addr, count)	Return <code>count</code> bytes as a string.
mem.write_block(addr, data)	Write the bytes of <code>data</code> .
mem.fill(addr, count, byte)	Fill a range with one byte.

If a raw RAM read or write is attempted while the CPU is not frozen, the script raises an error. Use `cpu.freeze()` before the access and `cpu.resume()` afterwards.

34.6 Terminal Module

term drives keyboard, mouse, and terminal text:

Function	Purpose
term.type(text)	Type text into the terminal.
term.type_line(text)	Type text followed by Return.
term.read()	Read pending terminal output.
term.clear()	Clear terminal output.
term.echo(on)	Enable or disable local echo.
term.wait_output(text, timeout)	Wait for text to appear.
term.mouse_move(x, y)	Move the mouse pointer.
term.mouse_delta(dx, dy)	Move the mouse pointer by a delta.
term.mouse_click(button)	Press and release a mouse button.
term.mouse_release(button)	Release a mouse button.
term.scancode(code)	Inject a keyboard scancode.
term.key_press(code)	Press and release a key.

34.7 Audio Module

audio controls the mixer and supported playback engines:

Function group	Purpose
<code>audio.start()</code> , <code>audio.stop()</code> , <code>audio.reset()</code>	Control audio output.
<code>audio.freeze()</code> , <code>audio.resume()</code>	Pause and resume audio processing.
<code>audio.write_reg(addr, value)</code>	Write an audio MMIO register.
<code>audio.set_master_gain_db(v)</code>	Set master gain in dB.
<code>audio.get_master_gain_db()</code>	Read master gain in dB.
<code>audio.set_master_auto_level_enabled(on)</code>	Enable automatic master levelling.
<code>audio.set_master_compressor_enabled(on)</code>	Enable master compression.
<code>audio.psg_load/play/stop/is_playing/metadata</code>	PSG playback helpers.
<code>audio.sid_load/play/stop/is_playing/metadata</code>	SID playback helpers.
<code>audio.ted_load/play/stop/is_playing</code>	TED playback helpers.
<code>audio.pokey_load/play/stop/is_playing</code>	POKEY playback helpers.
<code>audio.ahx_load/play/stop/is_playing</code>	AHX playback helpers.
<code>audio.midi_load/play/stop/pause/resume/set_volume/is_playing/metadata</code>	MIDI/MUS playback helpers.

Live MIDI has no separate high-level script helper. A script that deliberately wants to drive it can use `audio.write_reg(0xF0BF4, byte)` for data bytes and `audio.write_reg(0xF0BF6, 1)` for reset, then use `dbg.io("midilive")` to inspect the port.

34.8 Video Module

video controls display chips, blitter operations, and frame inspection:

Function group	Purpose
<code>video.write_reg(addr, value)</code> , <code>video.read_reg(addr)</code>	Raw video MMIO.
<code>video.get_dimensions()</code> , <code>video.is_enabled()</code>	Current VideoChip state.
<code>video.vga_enable(on)</code> , <code>video.vga_set_mode(mode)</code>	VGA control.
<code>video.vga_set_palette(i, r, g, b)</code>	VGA palette write.
<code>video.ula_enable(on)</code> , <code>video.ula_border(n)</code>	ULA control.
<code>video.antic_enable(on)</code> , <code>video.antic_dlist(addr)</code>	ANTIC control.
<code>video.gtia_color(i, value)</code>	GTIA colour register write.
<code>video.ted_enable(on)</code> , <code>video.ted_mode(a, b)</code>	TED control.
<code>video.voodoo_enable(on)</code> , <code>video.voodoo_draw()</code>	Voodoo control.
<code>video.copper_enable(on)</code> , <code>video.copper_set_program(addr)</code>	Copper control.
<code>video.blit_copy(...)</code> , <code>video.blit_fill(...)</code> , <code>video.blit_line(...)</code>	Blitter commands.
<code>video.blit_wait()</code>	Wait until the blitter is idle.
<code>video.get_pixel(x, y)</code>	Return one composited RGBA pixel.
<code>video.get_region(x, y, w, h)</code>	Return a rectangle of composited RGBA bytes.

Function group	Purpose
<code>video.frame_hash()</code>	Hash the current frame.
<code>video.wait_pixel(...)</code>	Wait for one pixel to match.
<code>video.wait_stable(frames, timeout)</code>	Wait for a stable frame hash.
<code>video.wait_condition(fn, timeout)</code>	Wait until callback <code>fn</code> returns true.

34.9 Recording Module

`rec` captures frames:

Function	Purpose
<code>rec.screenshot(name)</code>	Save one frame.
<code>rec.start(name)</code>	Start recording.
<code>rec.start_screen(name)</code>	Start screen recording.
<code>rec.stop()</code>	Stop and finalise recording.
<code>rec.is_recording()</code>	Return true while recording.
<code>rec.frame_count()</code>	Number of recorded frames.

34.10 Debug Module

`dbg` drives IE Mon from a script:

Function group	Purpose
<code>dbg.open()</code> , <code>dbg.close()</code> , <code>dbg.is_open()</code>	Control monitor visibility.
<code>dbg.step()</code> , <code>dbg.continue()</code> , <code>dbg.run_until(addr)</code>	Execution control.
<code>dbg.set_bp(addr)</code> , <code>dbg.clear_bp(addr)</code> , <code>dbg.list_bp()</code>	Breakpoints.
<code>dbg.set_wp(addr)</code> , <code>dbg.clear_wp(addr)</code> , <code>dbg.list_wp()</code>	Watchpoints.
<code>dbg.get_reg(name)</code> , <code>dbg.set_reg(name, value)</code>	Register access.
<code>dbg.get_pc()</code> , <code>dbg.set_pc(addr)</code>	Program counter access.
<code>dbg.read_mem(addr, n)</code> , <code>dbg.write_mem(addr, data)</code>	Memory access through the monitor.
<code>dbg.disasm(addr, count)</code>	Disassemble instructions.
<code>dbg.backtrace()</code>	Return a stack backtrace.
<code>dbg.backtrace_frames(depth)</code>	Return structured backtrace frames.
<code>dbg.timeline(count)</code>	Return recent timeline entries.
<code>dbg.io_devices()</code> , <code>dbg.io(name)</code>	Inspect IE Mon I/O register views.
<code>dbg.history_horizon()</code> , <code>dbg.history_config(opts)</code>	Inspect or configure whole-machine reverse-history retention.

Function group	Purpose
<code>dbg.tracing_on(size)</code> , <code>dbg.tracing_off()</code> , <code>dbg.tracing_show(count)</code>	Control the focussed CPU trace ring.
<code>dbg.device_list()</code> , <code>dbg.device_snapshot(name)</code> , <code>dbg.device_diff(a,b)</code>	Inspect versioned device snapshots.
<code>dbg.save_state(path)</code> , <code>dbg.load_state(path)</code>	Save or load a CPU-local monitor snapshot.
<code>dbg.on_fault(kind, fn)</code>	Call <code>fn</code> when a selected fault occurs.
<code>dbg.poll_faults()</code>	Poll pending fault events.
<code>dbg.command(line)</code>	Run one IE Mon command.

Fault callbacks receive a table with `cpu_id`, `pc`, `addr`, `kind`, and `info` fields.

The debug module mirrors the monitor where scripts need repeatable inspection. `dbg.io_devices()` returns the monitor's named I/O register views, and `dbg.io(name)` returns one table entry per register with `name`, `addr`, `value`, and `access` fields. The values use the same native-width MMIO read path as IE Mon `io`, so a script inspecting a long register gets a long register value even when the focussed CPU is a narrow bus client. An unknown view name returns an empty table rather than raising an error; check `dbg.io_devices()` before relying on a view name. MIDI has two useful views: `midisplay` for the file player and `midilive` for the byte stream port.

Trace-ring helpers return structured recent-instruction entries, `dbg.backtrace_frames()` returns one table per call frame, and the history helpers report or configure the whole-machine reverse timeline used by IE Mon `rg` and `rt`. Device helpers snapshot registered versioned devices and compare two snapshots without forcing the script to parse monitor text.

`dbg.save_state` and `dbg.load_state` use IE Mon `ss` and `sl`, so they are CPU-local snapshots. They are not whole-machine save files and do not include other CPUs, device state, audio/video state, timers, DMA, or reverse-history retention.

34.11 Symbols, Regions, and Bits

Module	Useful functions
<code>sym</code>	<code>add</code> , <code>lookup</code> , <code>resolve</code> , <code>list</code>
<code>regions</code>	<code>list</code> , <code>lookup</code>
<code>bit32</code>	<code>band</code> , <code>bor</code> , <code>bxor</code> , <code>bnot</code> , <code>lshift</code> , <code>rshift</code> , <code>arshift</code> , <code>lrotate</code> , <code>rrotate</code> , <code>btest</code> , <code>extract</code> , <code>replace</code>

34.12 Runnable Video Example

Store this as `FIRST.IES`, then run `RUN "FIRST.IES"` from BASIC or `script FIRST.IES` from IE Mon:

```
-- Draw a blue VideoChip field with a green diagonal.
sys.print("IE SCRIPT VIDEO")
video.write_reg(983044, 4)
video.write_reg(983172, 1048576)
video.write_reg(983040, 1)
video.blit_fill(1048576, 320, 200, 255, 1280)
video.blit_line(0, 0, 319, 199, 65280)
video.blit_wait()
sys.print("BLT " .. video.read_reg(983108))
sys.quit()
```

The comment marks the visible job before the device writes begin. The script sets VideoChip mode 4 (320 by 200), selects framebuffer base \$100000, enables the chip, fills the framebuffer, draws a diagonal, waits for the blitter, and prints the blitter status. The expected status is BLT 2, meaning DONE set and ERR clear. Try changing 65280 to 16711680 in the `video.blit_line` call; the diagonal changes from green to red.

34.13 Runnable Audio Example

This companion script uses the same bus-facing style for sound. It programs SoundChip channel 0 through the flexible-channel registers, waits for a short time, and prints back the control register so you can see the gate bit that was written.

Store this as `TONE.IES`:

```
-- SoundChip channel 0, square wave at about middle C.
audio.start()
audio.write_reg(0xF0A80, 262 * 256)
audio.write_reg(0xF0A84, 96)
audio.write_reg(0xF0AA4, 0)
audio.write_reg(0xF0A88, 3)
sys.wait_ms(250)
sys.print("CH0 " .. mem.read32(0xF0A88))
sys.quit()
```

`audio.start()` enables the global audio path. The frequency register uses 16.8 fixed-point hertz, so `262 * 256` means about 262 Hz. The volume write sets an audible level, `WAVE_TYPE 0` selects a square wave, and control value 3 means enabled plus gate. The print should include `CH0 3`.

34.14 Fault Callback Example

This pattern records the program counter for any IE64 illegal instruction fault and then exits cleanly after a short wait:

```
local seen = 0

dbg.on_fault("ie64.illegal", function(ev)
    seen = ev.pc
    sys.print("FAULT PC " .. seen)
end)

sys.wait_ms(100)
sys.quit()
```

34.15 Limits and Error Behaviour

Scripts run cooperatively. A script that loops forever without calling a wait function cannot be cancelled promptly.

Storage helpers are limited to approved script storage. Names that escape that storage are rejected before any read or write occurs.

Raw RAM access through `mem` requires `cpu.freeze()`. MMIO access does not. If a script fails after freezing a CPU, audio, or the monitor, IE Script releases those holds before returning control.

34.16 What Comes Next

Part IV ends here. Part V covers persistent storage, machine control commands, input MMIO, and the serial interface.