

ngspice-jit — command reference

Reference manual for **ngspice-jit**, a fork of **ngspice** (ngspice-46 base) that JIT-compiles behavioral foundry R/C into native OSDI devices at parse time. A command-by-command reference with full `cp_coms` catalog coverage; every example is executed against the ngspice-jit build and its real output shown verbatim. Fork-specific features (OSDI codegen, regex device/vector selection, the `devices` command) are flagged as build extensions where they appear.

Contents

- 1. General netlist & language syntax — *reserved (written last)*
- 2. Key concepts & gotchas — batch vs interactive, echo vs print, plot vs .plot, namespaces, parse-time
- 3. Command reference (A–Z) — every dot-card and control command, alphabetical
- 4. OSDI — compiling behavioral foundry passives to native devices

Alphabetical command index

`.ac` — small-signal AC (frequency) analysis

A device / `codemodel` — XSPICE code models

`alias` / `unalias` — command shortcuts

`alter` — change a device value between runs

`altermod` — change a model parameter between runs

`@device[param]` — read a device/model parameter

`compose` — build a vector

`.csparam` — expose a parameter to `.control`

`.dc` — DC sweep

`define` — user functions

`devhelp` / `devices` / `show` / `showmod` — device & model introspection

`diff` — compare two plots

`display` — list vectors in the current plot

`.disto` — small-signal distortion analysis

`echo` — print text

`fft` — FFT of a transient vector

`fopen` / `fread` / `fclose` — read a file line by line

`foreach` — iterate over a word list

`.four` / `fourier` — Fourier analysis of a transient

`.global` — global nodes

`goto` / `label` — control-flow jump

`.ic` / `.nodeset` — initial node conditions

`.if` / `.elseif` / `.else` / `.endif` — conditional netlist

`.include` / `.lib` — pull in other files

`let` / `print` — vectors and expressions

`linearize` — resample onto a uniform time grid

`listing` — show the parsed circuit

`load` — read a raw file back

`.meas` / `meas` — extract figures of merit

`.model` — named device model

`.noise` — noise analysis

`.op` — DC operating point

`.option` / `.options` — simulator options

Other commands — less-common, alias & developer

`.param` / `.func` — parameters and functions

`.plot` / `plot` — a curve, ASCII or graphical

`print` / `.print` — values as text

`.pz` — pole-zero analysis

`repeat` / `if` / `else` — fixed count & branching

`reset` / `rehash` / `help` / `quit` — session commands

`reshape` / `transpose` / `cutout` / `cross` — vector surgery

`.save` / `save` — choose what is stored

`.sens` — sensitivity analysis

`set / $ / $&` — shell variables vs. vector values
`setplot / setscale / destroy` — manage plots
`setseed` — seed the random generator
`settype` — set a vector's type/units
`shell / cd / getcwd` — OS shell & directory
`snsave / snload` — save/restore simulation state
`source / edit` — load a deck interactively
`.sp / .pss / .hb` — RF & large-signal analyses
`spec` — spectrum over a frequency range
`.step` — not supported (use a loop)
`stop / resume / delete` — breakpoints
`strcmp / strstr / strslice` — string operations
`.subckt / .ends / X` — subcircuits
`.temp` — simulation temperature
`.tf` — DC small-signal transfer function
`.tran` — transient analysis
`unlet` — delete vectors
`version / sysinfo / rusage / history` — session info
`where / dump / state / status` — diagnostics
`while / dowhile` — condition loops
`wrdata` — ASCII column data
`write` vs. `-r` — two ways to make a raw file
`osdi` — compiled foundry passives

Grouped by topic below while drafting; the dot-cards will be re-sorted into a single A–Z reference in the final pass.

Notation. Each entry carries a mode badge: the command works both in batch (`ngspice -b`) and interactively; it exists only inside `.control ... .endc` or the interactive prompt. The badge is taken directly from the `spiceonly` field of the command table in the source, so it is authoritative.

1. General netlist & language syntax

This chapter assumes you know what a SPICE netlist is; it is the reference for ngspice’s specific syntax rules — the ones that bite. Individual commands are in §3.

Deck structure

A deck is a **title line**, a body of cards, and `.end` :

- **The first line is always the title** and is ignored — even if it looks like a valid card. Putting a real card (e.g. `.save`) on line 1 silently does nothing. Start every deck with a comment or title line.
- **Comments:** a line starting with `*` is a full-line comment; `$` (with a preceding space) starts an **in-line** comment to end of line.
- **Line continuation:** a line beginning with `+` is appended to the previous card.
- **Case-insensitive:** card names, node names, and parameters are not case-sensitive (`VDD = vdd`).
- `.end` terminates the deck; text after it is ignored.

Element cards

Every instance card begins with a **letter that selects the device type**, followed by the instance name, its terminal nodes, and a value or model plus parameters:

Rload	out 0	1k	\$ R + name; nodes out,0; value 1k
M1	d g s b	nch w=1u l=0.18u	\$ M + name; 4 nodes; model; params
X1	in out	amp	\$ X = subcircuit call (see .subckt)

letter	device	letter	device
R C L	resistor, capacitor, inductor (<i>K</i> =mutual)	V I	independent voltage / current source
D Q	diode, BJT	M J Z	MOSFET, JFET, MESFET
E G	VCVS, VCCS (linear controlled)	F H	CCCS, CCVS
B	arbitrary (nonlinear) source $V= / I=$	X	subcircuit call → <code>.subckt</code>
S W	voltage / current switch	A	XSPICE code model

Nodes

Nodes are named or numbered; **0 (or gnd) is ground** and is global. Names are case-insensitive. A node reachable everywhere without wiring is declared with `.global` ; a node inside a subcircuit instance is addressed hierarchically as `Xinst.node` (see `.subckt`).

Numbers & unit suffixes

A number may carry a scale suffix (case-insensitive); trailing letters after the suffix are ignored (`1k0hm` = `1k`). **The classic trap: *m* is *milli*, not *mega* — mega is *meg* .**

T	G	Meg	k	m	u	n	p	f	a
1e12	1e9	1e6	1e3	1e-3	1e-6	1e-9	1e-12	1e-15	1e-18

syntax.cir
— run: `ngspice -b syntax.cir (meg vs m, and + continuation)`

```

* first line is the TITLE and is ignored
.options savecurrents
V1 in 0 DC 1
R1 in 0 1meg      $ 1,000,000 ohm  -> I = 1 uA
R2 in 0 1k        $ 1,000 ohm     -> I = 1 mA
R3 in 0 1m        $ 0.001 ohm    (m = MILLI!)  -> I = 1000 A
Rcont in 0
+ 2k              $ line continued with '+'  -> I = 0.5 mA
.op
.control
  run
  print @R1[i] @R2[i] @R3[i] @Rcont[i]
.endc
.end

```

→ output

```

@r1[i] = 1.000000e-06      (1 V / 1 MΩ)
@r2[i] = 1.000000e-03      (1 V / 1 kΩ)
@r3[i] = 1.000000e+03      (1 V / 1 mΩ ← m is milli!)
@rcont[i] = 5.000000e-04   (1 V / 2 kΩ, value read off the + line)

```

Gotcha — *m* vs *meg* .
A “1 mΩ” typo for a 1 MΩ resistor (`1m` instead of `1meg`) is off by **10⁹** — the run above draws 1000 A instead of 1 μA. This is the single most common SPICE numeric error.

Parameter expressions

Expressions in a value are wrapped in `{ ... }` or `' ... '` and are evaluated **at parse time** (before the run) over `.param` / `.func` values and constants — they cannot see simulation results. Run-time math on vectors is done instead with `let` / `define` inside `.control` (see §2).

Two languages in one file

A deck mixes two sub-languages: **dot-cards** (the netlist — `.tran` , `.model` , `.param` , ...) processed by the parser, and **commands** inside a `.control ... .endc` block (`run` , `let` , `print` , loops) executed by the interpreter. Which one runs a given verb, and when, is the source of most surprises — covered in §2 · `.control` vs. a pure dot-card deck (why ngspice differs from Eldo/HSPICE, when you need `.control` , and what works without it).

2. Key concepts & gotchas

Cross-cutting behaviors that trip people up; command entries link back here.

Batch mode needs an output directive. In batch (`ngspice -b`), an analysis card only runs if the deck also produces output — a `.control { run ... }` block, or a `.print` / `.plot` / `.four` line. A deck with just `.pz` (or `.sens` , `.ac` , ...) and nothing else prints “*no .plot, .print, or .fourier lines in batch mode; no simulations run!*” and exits without computing anything. `.op` , `.tf` and `.four` are exceptions — they emit their result directly. Interactively (or with `-r`) this restriction does not apply.

Which nodes end up in the raw file, and whether a file is even produced, depends on **three interacting choices**: the `-r` command-line flag, whether you used `.save` / `save` , and whether a `.control` block calls `write` . The table is the mental model; the examples verify it.

command	mode	behavior
<code>.save node ...</code>	<code>batch + ui</code>	restricts the <code>-r</code> raw file to the listed vectors (plus the <code>time</code> scale)
<code>save node ...</code>	<code>batch + ui</code>	same restriction, issued as a command (e.g. inside <code>.control</code>)
<code>write file expr ...</code>	<code>ui only</code>	writes selected vectors to <i>its own</i> file; has no dot-card form
<code>.probe</code> / <code>probe</code>	batch	adds device currents to the raw file; no-op without <code>-r</code>

Inside `.control` , ngspice has two separate namespaces that look alike but are not: **vectors** (numeric simulation data — nodes, currents, anything you compute with `let`) and **shell variables** (text/flags set with `set`). The `$` / `&` sigils and the `echo` / `print` split all come down to which namespace you mean.

The one rule that removes most confusion: `echo` is literal, `print` evaluates. `echo` prints its words verbatim (expanding only `$$shellvar`); it will happily print `v(out)` or `@R1[resistance]` as plain text. `print` evaluates them as vectors/expressions. If a value comes out as its own name, you used `echo` where you needed `print` .

These are **control-mode** constructs — they live inside a `.control ... .endc` block (or the interactive prompt), never as dot-cards. They turn a netlist into a small script: sweep a value, retry until convergence, branch on a result. Each loop body is closed by `end` . They rely on the shell variables and vectors covered in the Vectors chapter (`let` , `set` , `$var` , `&$vec`).

Loop vs. sweep. A `foreach` / `while` loop that re-runs an analysis is *not* the same as `.dc` / `.step` : the loop runs a full separate analysis each pass (you control saving/plotting), whereas a sweep produces one dataset indexed by the swept variable. Use a loop when each pass needs different *topology/parameters* or its own output file; use a sweep for one parametric curve.

Commands that load, run, and modify the circuit between runs — the basis of interactive what-if sweeps without re-parsing the deck. `run` starts the defined analysis; `resume` continues a paused one; `reset` reloads the circuit to its initial state; `source file` loads another deck. The two most useful for scripting are `alter` and `altermod` .

These dot-cards give a netlist its structure: named parameters, reusable functions, hierarchical subcircuits, and file inclusion. Parameter/function expressions are wrapped in `{ ... }` (or `' ... '`) and are evaluated **at parse time**, before the simulation runs.

.controlendc vs. a pure dot-card deck — why ngspice has two languages. A single ngspice file mixes **two** languages, and it helps to know which one you are writing in. This is the biggest structural difference from Eldo and HSPICE, which use one unified deck language and push scripting/plotting into dedicated dot-cards (`.STEP` , `.ALTER` , `.MEASURE`) and an external waveform viewer.

ngspice descends from Berkeley **SPICE3 + nutmeg**: the *simulator* reads a declarative netlist, while *nutmeg* is an interactive command shell (type `let` , `plot` , `foreach` ... at the `ngspice ->` prompt). The `.control ... .endc` block is simply “**run these nutmeg commands automatically**” embedded in the file — an imperative script that runs after the circuit is parsed.

	netlist / dot-card language	control (nutmeg) language
style	declarative — a circuit + what to analyze	imperative — a script, run top to bottom
examples	<code>.tran</code> <code>.ac</code> <code>.dc</code> <code>.model</code> <code>.param</code> <code>.print</code> <code>.save</code>	<code>run</code> <code>let</code> <code>foreach</code> <code>while</code> <code>if</code> <code>alter</code> <code>meas</code> <code>fft</code> <code>write</code> <code>plot</code>
runs	during the simulator pass	after parse, on demand, inside <code>.controlendc</code>

Can you do everything without `.control` ? Basic simulation, yes; the full tool, no — the whole scripting/post-processing half of ngspice lives only inside `.control` . ngspice deliberately ships *both* forms of several features so HSPICE-style batch decks work unchanged (`.four` card / `fourier` command, `.plot` ASCII card / graphical `plot` , `.meas` card / `meas` command) — that

duplication is the compatibility bridge.

works as pure dot-cards (batch -b , no .control)	only available inside .control
analyses .op .tran .ac .dc .noise .tf .disto .pz .sens .four ; output .print .plot .save .meas .options .param .temp .model .subckt .include .ic ; rawfile via -r , log via -o	scripting let set foreach while if repeat ; alter / altermod between runs ; vector math fft spec psd linearize load ; chosen-set write / wrdata ; graphical plot / hardcopy / gnuplot ; diff ; sequenced multi-analysis ; Monte Carlo (loop + mc_source)

Rule of thumb. A plain dot-card deck behaves like HSPICE batch mode — reach for it when one fixed analysis with standard output is all you need. Add `.control` the moment you need loops, logic, several analyses in sequence, or post-processing beyond `.print` / `.plot`. Two traps at the boundary: a `.param` is **not** visible inside `.control` (promote it with `.csparam`), and batch mode still needs an output directive (`.print` / `.plot` / `.four` or `-r`) or nothing runs. To accept more HSPICE deck syntax overall, set `ngbehavior=hs` in `spinit` / `.spiceinit`.

.spiceinit / **spinit** — the startup files. ngspice runs a startup script at launch: a list of **front-end commands** executed as if typed at the `ngspice ->` prompt. It is **not** a netlist — there is no circuit yet, so device cards and analysis dot-cards do not belong here, and no `.control ... .endc` wrapper is needed (the file is already in command context; comment lines start with `*`). Two files share the same syntax: **spinit** is system-wide (in the install/share dir — it loads code models and sets machine defaults), and **.spiceinit** is yours. ngspice searches for `.spiceinit` in the **current directory first, then your home directory**, so a project can carry its own. Both are sourced on every start, batch or interactive.

It configures **the tool**; the deck describes **the circuit**. Anything valid as a front-end command works — `set / unset` , `option` , `alias` , `codemodel` / `osdi` / `use` , `if/end` :

category	examples	what they do
dialect	<code>set ngbehavior=hs</code> (also <code>ps</code> / <code>lt</code> / <code>hsa</code> / <code>psa</code> / <code>lta</code>)	accept HSPICE / PSPICE / LTspice deck syntax
performance	<code>set num_threads=8</code> · <code>option klu</code>	OpenMP thread count · pick the KLU solver as default
output defaults	<code>set filetype=ascii</code> · <code>set hcopypscolor=1</code> · <code>option noinit</code>	ASCII rawfiles · colour hardcopy · suppress the OP dump
option defaults	<code>option reltol=1e-4</code> · <code>set nostepsize-limit</code> · <code>set ng_nomodcheck</code>	tolerances / behavior applied to circuits loaded afterward
aliases	<code>alias exit quit</code> · <code>alias acct rusage all</code>	command shortcuts
load models / libs	<code>codemodel ../analog.cm</code> · <code>osdi ../BSIMCMG.osdi</code> · <code>use lib</code>	load XSPICE code models / OSDI compiled models / device libraries
enable / disable OSDI	<code>set osdi_enabled</code> · <code>unset osdi_enabled</code>	the switch the shipped <code>spinit</code> tests to load (or skip) the <code>.osdi</code> model libraries
front-end vars	<code>set rndseed=12</code> · <code>set askquit</code> · <code>set ngdebug</code> · <code>set sourcepath=(...)</code>	fix the RNG seed · confirm on quit · debug trace · <code>source</code> / <code>.include</code> search path
control flow	<code>if \$?xspice_enabled ... end</code> · <code>unset osdi_enabled</code>	conditional setup (the shipped <code>spinit</code> guards code-model loading this way)

What must not go in it. Circuit content — device lines, `.model` , `.subckt` , `.param` , and analysis cards (`.tran` / `.ac` / `.dc`) — has no circuit to attach to at startup; keep it in the deck. Use the `option ... command` form here, not the `.options` dot-card. And don't wrap the file in `.control ... .endc` : it is redundant. Verified: a `set myflag=42` in `./spiceinit` is readable as `$myflag` inside a later deck's `.control`.

Turning OSDI on and off. The shipped `spinit` wraps its model loads in a guard — `if $?osdi_enabled ... osdi lib1.osdi ... end` — so a single variable decides whether the compiled `.osdi` libraries load. On this build `osdi_enabled` is **not** defined by default (`$?osdi_enabled` = 0), so OSDI loading is **opt-in**: add `set osdi_enabled` before the block to enable it, or `unset osdi_enabled` (the shipped default) to skip it. The XSPICE code models work the same way via `$?xspice_enabled` . Two caveats: the variable is only a `spinit` convention — a bare `osdi foo.osdi` line loads that library regardless — and this switch governs *loading external OSDI/OpenVAF model libraries*, which is distinct from the parse-time §4 behavioral-passive codegen (that one triggers on behavioral `R=` / `C=` expressions in the netlist and has its **own** opt-out, `set osdi_res_bsource` — also placed in `.spiceinit` because it acts at parse time).

3. Command reference (A-Z)

Every dot-card and control command in one alphabetical sequence (leading dot ignored for ordering). See the index above to jump.

.ac

— small-signal AC (frequency) analysis

batch + ui

SYNOPSIS

```
.ac dec|oct|lin n fstart fstop
```

DESCRIPTION

Linearizes the circuit about its DC operating point and sweeps frequency, giving complex node voltages (magnitude/phase). Stimulus comes from sources carrying an AC value.

PARAMETERS

parameter	meaning
dec / oct / lin	points spaced per decade / per octave / linearly
n	points per decade/octave, or total points for lin
fstart fstop	frequency range (Hz)

EXAMPLE

— RC low-pass corner

ac.cir — run: ngspice -b ac.cir

```
* RC low-pass AC magnitude, find -3 dB corner
V1 in 0 AC 1
R1 in out 1k
C1 out 0 100n
.ac dec 50 10 1meg
.control
  run
  meas ac fc WHEN vdb(out)=-3.0103
.endc
.end
```

→ output

fc = 1.59155e+03

$fc = 1/(2\pi \cdot R \cdot C) = 1/(2\pi \cdot 1\text{ k}\Omega \cdot 100\text{ nF}) = 1591.55\text{ Hz}$ ✓. Note vdb() gives magnitude in dB; -3.0103 dB is the half-power point.

EXAMPLE

— phase crosses −45° at the corner

ac_phase.cir — run: ngspice -b ac_phase.cir

```
* phase = -45 deg (= -pi/4 rad) exactly at the corner
V1 in 0 AC 1
R1 in out 1k
C1 out 0 100n
.ac dec 50 10 1meg
.control
  run
  meas ac f45 WHEN vp(out)=-0.7853982 $ vp() is in RADIANS
.endc
.end
```

→ output

f45 = 1.59169e+03

Same 1592 Hz corner, cross-checked through the phase instead of the magnitude ✓.

Gotcha — vp() returns radians, not degrees. A single-pole low-pass is −45° at its corner, but meas ... WHEN vp(out)=-45 fails with out of interval because vp() ranges only −π/2...0. Use the radian value -0.7853982 (= −π/4), or take cph() for continuous (unwrapped) phase.

Gotcha — no AC value, no stimulus. `.ac` only injects signal through sources given an `AC` magnitude (e.g. `V1 in 0 AC 1`). A source with only a `DC` or transient (`SIN` / `PULSE`) spec contributes nothing in AC — the output is silently zero.

SEE ALSO

`.noise` , `.tf` , `.op` .

A device / `codemodel` — XSPICE code models

batch + ui

SYNOPSIS

```
Aname in-ports ... out-ports ... modelname
.model modelname codemodel_type ( params )
codemodel library.cm                $ load an external code-model library
```

DESCRIPTION

The `A` card instantiates an XSPICE **code model** — a compiled behavioral block (gain, summer, limiter, s-domain transfer function, ADC/DAC, a file-driven source, ...) beyond the built-in analog primitives. The code models are **not linked into ngspice**; they live in shared-library files (`analog.cm` , `digital.cm` , `xtrdev.cm` , ...) and are loaded at run time by the `codemodel` command. In a normal install `spinit` already issues those `codemodel` lines at start-up (guarded by `$? xspice_enabled`), so a deck can use `gain` , `filesource` , etc. with **no `codemodel` line of its own**.

`amodel.cir` — a gain block; run: `ngspice -b amodel.cir`

```
V1 in 0 DC 0.5
A1 in out gainblk
.model gainblk gain(gain=3.0)
.control
  op
  print v(out)      $ 3.0 * 0.5 = 1.5
.endc
.end
```

→ output
v(out) = 1.500000e+00

EXAMPLE

— a

VECTOR

pattern generator (`filesource` driving a whole bus)

`buspattern.cir` + `pattern4.dat` — run: `ngspice -b buspattern.cir`

```
* one A-device drives a 4-bit bus from a multi-column file: a binary counter
a1 %v([b3 b2 b1 b0]) bus4          $ a VECTOR port -> four nodes at once
.model bus4 filesource (file="pattern4.dat"
+ amploffset=[0 0 0 0] amplscale=[1.8 1.8 1.8 1.8]  $ per-bit: logic 1 = 1.8 V
+ timeoffset=0 timescale=1 timerelative=false amplstep=TRUE)
rb3 b3 0 1meg
rb2 b2 0 1meg
rb1 b1 0 1meg
rb0 b0 0 1meg
.tran 5n 1.5u
.control
  run
  meas tran b3_7  FIND v(b3) AT=750n      $ counter=7  (0111): b3=0,   b0=1.8
  meas tran b0_7  FIND v(b0) AT=750n
  meas tran b3_12 FIND v(b3) AT=1250n     $ counter=12 (1100): b3=1.8, b0=0
  meas tran b0_12 FIND v(b0) AT=1250n
.endc
.end
```

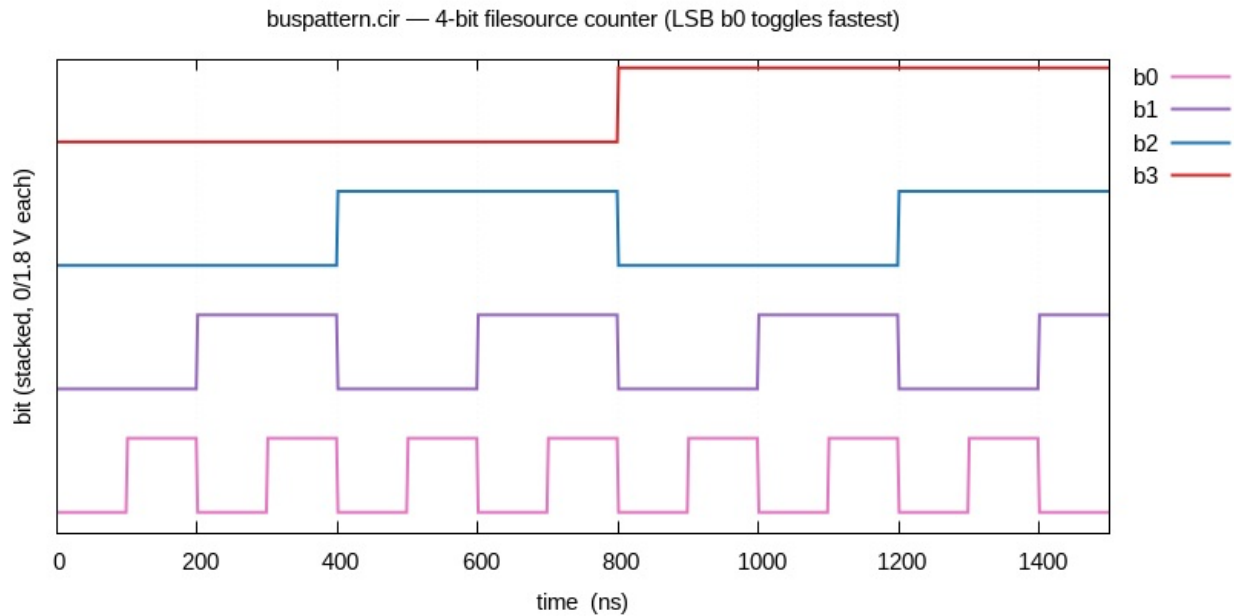
→ output

b3_7	=	0.000000e+00
b0_7	=	1.800000e+00
b3_12	=	1.800000e+00
b0_12	=	0.000000e+00

This is where code models earn their keep. A **vector** port `%v([b3 b2 b1 b0])` lets one `filesource` drive a whole bus: each column of `pattern4.dat` feeds one bit, scaled independently by the per-port `amplscale` / `amploffset` vectors — here a 4-bit binary counter with logic-1 = 1.8 V. `amplstep=TRUE` holds each level (clean digital edges); `false` would linearly interpolate (analog PWL). At t = 750 ns the file’s row is `0 1 1 1` and t = 1250 ns is `1 1 0 0` — both read back exactly ✓. Ports carry a

type sigil: `%v([n])` single-ended, `%vd([p n])` differential, `%i` current, `%vnam` named. A fuller version — differential sine/cosine plus an 8-bit bus — is at `/users/design/ngspice/code_model_pattern.sp`.

The four bits, stacked — the classic binary cascade (`b0` toggles every 100 ns, each higher bit half as often). `bus4.gp`: `wrdata bus4.txt v(b3)..v(b0)` then `gnuplot bus4.gp`.



EXAMPLE
— a quadrature (cos/sin) generator via a
DIFFERENTIAL
port

`sincos.cir` + `sincos.dat` — run: `ngspice -b sincos.cir`

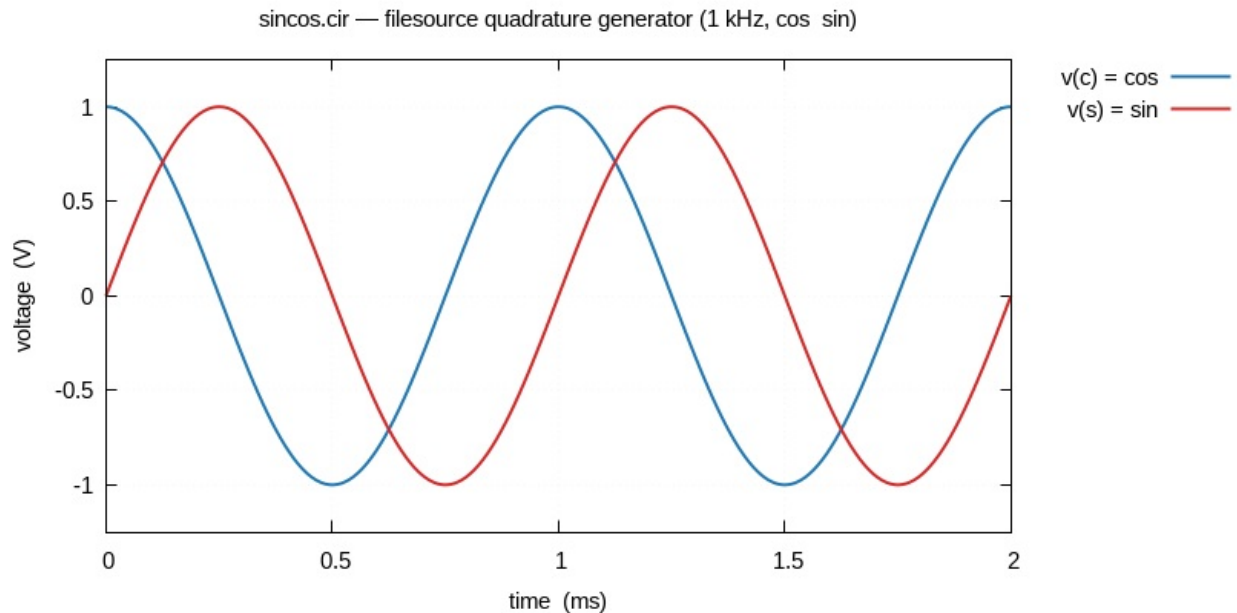
```
* two quadrature channels from a 3-column file (time, cos, sin)
a1 %vd([c 0 s 0]) sincos          $ %vd pairs nodes -> differential ports
.model sincos filesource (file="sincos.dat"
+ amploffset=[0 0] amplscale=[1 1]
+ timeoffset=0 timescale=1 timerelative=false amplstep=false)
rc c 0 1meg
rs s 0 1meg
.tran 2u 2m
.control
  run
  meas tran c_0 FIND v(c) AT=0      $ cos(0)   = 1
  meas tran s_0 FIND v(s) AT=0      $ sin(0)   = 0
  meas tran c_q FIND v(c) AT=250u   $ cos(90d) = 0
  meas tran s_q FIND v(s) AT=250u   $ sin(90d) = 1
.endc
.end
```

→ output

c_0	=	1.00000e+00	
s_0	=	0.00000e+00	
c_q	=	1.73472e-18	(= 0)
s_q	=	9.99920e-01	(= 1)

Here `%vd([c 0 s 0])` declares two **differential** output ports — one across (`c`,0), one across (`s`,0) — fed by columns 2 and 3 of the file. With `amplstep=false` the samples are linearly interpolated, giving smooth analog waveforms; the two channels are 90° apart (cos and sin) ✓. Because each port’s reference here is ground, the pair acts single-ended — give a non-zero second node to drive a genuinely floating differential output.

The two channels — $v(c)$ and $v(s)$ in quadrature (90° apart), smoothly interpolated from the file. `sincos.gp`: `wrdata sincos.txt v(c) v(s)` then `gnuplot sincos.gp`.



EXAMPLE

— an 8-bit bus with

PER-BIT OFFSETS

(stacked logic traces)

`bus8.cir` + `pattern8.dat` — run: `ngspice -b bus8.cir`

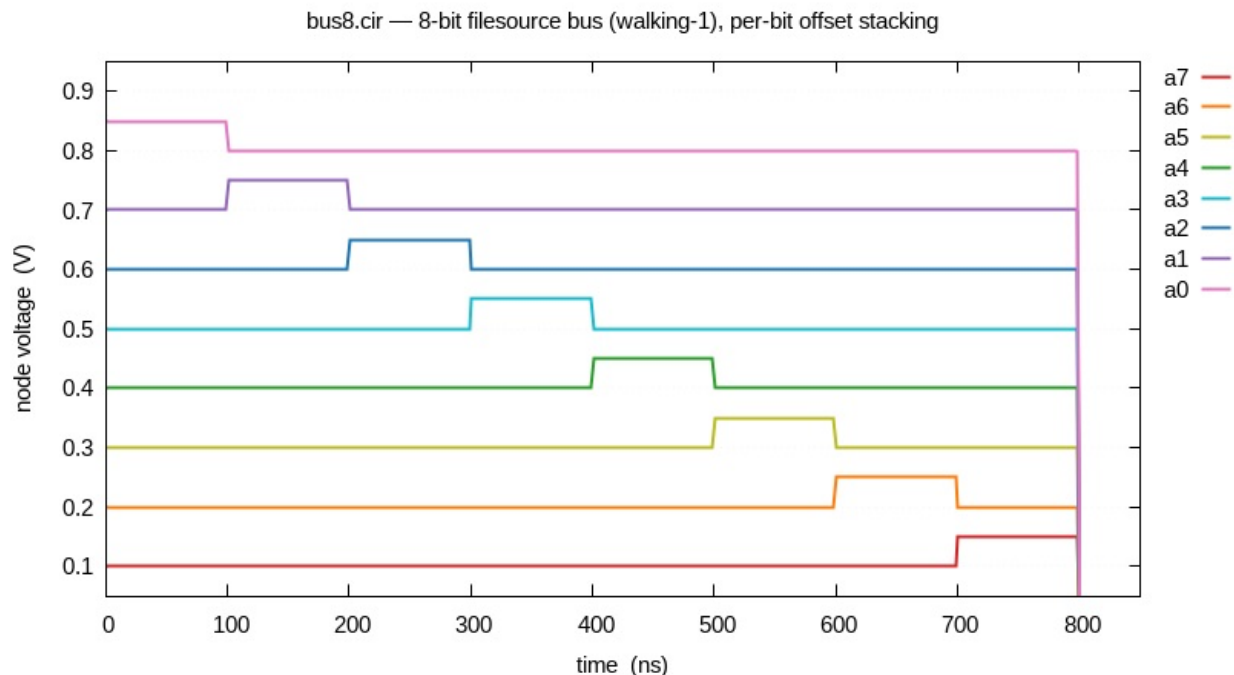
```
* 8 bits, each parked at a distinct DC level so the traces don't overlap
a1 %v([a7 a6 a5 a4 a3 a2 a1 a0]) bus8
.model bus8 filesource (file="pattern8.dat"
+ amploffset=[0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8]      $ per-bit DC level
+ amplscales=[0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05] $ small swing
+ timeoffset=0 timescale=1 timerelative=false amplstep=TRUE)
ra7 a7 0 1meg
ra0 a0 0 1meg
.tran 5n 850n
.control
  run
  meas tran a0_hi FIND v(a0) AT=50n      $ a0 hot: 0.8 + 0.05 = 0.85
  meas tran a7_lo FIND v(a7) AT=50n      $ a7 low: 0.1 + 0    = 0.10
  meas tran a7_hi FIND v(a7) AT=750n     $ a7 hot: 0.1 + 0.05 = 0.15
  meas tran a0_lo FIND v(a0) AT=750n     $ a0 low: 0.8 + 0    = 0.80
.endc
.end
```

→ output

```
a0_hi      = 8.50000e-01
a7_lo      = 1.00000e-01
a7_hi      = 1.50000e-01
a0_lo      = 8.00000e-01
```

The eight-column file is a walking-1; the per-bit `amploffset` vector (0.1...0.8 V) parks each bit at its own DC level while `ampscale` keeps the swing to 0.05 V, so plotting the bus shows eight non-overlapping stacked traces — the usual way to eyeball a digital vector. All four probes match their `offset + scale·bit` ✓.

The eight bus traces — the walking 1 climbs diagonally while each bit stays parked at its own level. Generated with `bus8.gp`: `wrdata bus8.txt v(a7)..v(a0) then gnuplot bus8.gp`.



Gotcha — filesource outputs 0 past the last sample. Beyond the final row's time the source drops to 0 (offsets included), not the last held value. Add a terminating row (here at 800 ns) if you need the pattern to persist to the end of the `.tran` — without it the `a7 / a0` probes at 750 ns read 0 instead of their levels.

CODEMODEL — LOADING A LIBRARY

You only call `codemodel path/lib.cm` yourself to load a **library that isn't already in** — typically a **custom, user-compiled .cm**, placed in `spinit / .spiceinit` (or a script sourced) *before* the circuit that uses it. That is why there is no standalone deck example here: the standard libraries are loaded for you at start-up, and the model just works.

Gotcha — don't re-load a library that's already loaded. Adding `codemodel ../analog.cm` to a deck when `spinit` already loaded it double-registers the models and **segfaults** this build. Load each `.cm` exactly once (leave the standard set to `spinit`); use an explicit `codemodel` only for a library not already present.

SEE ALSO

`.model`, `.spiceinit` (where `codemodel` normally lives), OSDI (compiled foundry passives, a different route).

SYNOPSIS

```
alias [ name [ definition ] ]      unalias name
```

DESCRIPTION

Defines a control-mode shortcut for a command and its arguments. `alias` with no arguments lists all aliases; `unalias` removes one.

↳ `alias.cir` — run: `ngspice -b alias.cir`

```
R1 1 0 1k
.control
  alias pp print
  let x = 6*7
  pp x           $ 'pp' now means 'print'
.endc
.end
```

→ output
x = 4.200000e+01

SEE ALSO

`define`.

SYNOPSIS

```
alter devname value
alter devname param = value
```

DESCRIPTION

Changes an instance parameter of an already-loaded device (a resistance, capacitance, source level, ...) without editing or re-parsing the netlist. Follow it with `run` to resimulate. Ideal inside a `foreach` / `while` loop for parameter studies.

`alter.cir` — run: `ngspice -b alter.cir`

```
V1 in 0 DC 1
R1 in out 1k
R2 out 0 1k
.tran 1u 5u
.control
  run
  print v(out)[0]      $ 0.5  (equal divider)
  alter R2 = 3k
  run
  print v(out)[0]      $ 0.75 (now a 1k:3k divider)
.endc
.end
```

→ output
v(out)[0] = 5.000000e-01
v(out)[0] = 7.500000e-01

EXAMPLE

— `alterparam` : change a `.param`, not a device

`alterparam.cir` — run: `ngspice -b alterparam.cir`

```
.param rval=1k
V1 in 0 DC 1
R1 in 0 {rval}
.op
.control
  run
  print @r1[resistance]
  alterparam rval=2k
  mc_source          $ rebuild the deck so the new .param takes effect
  op
  print @r1[resistance]
.endc
.end
```

→ output
@r1[resistance] = 1.000000e+03
@r1[resistance] = 2.000000e+03

Where `alter` / `altermod` poke a device or model directly, `alterparam` edits a `.param` value. Because parameters are resolved when the netlist is expanded, it only takes effect after a re-expansion — `mc_source` (or `reset`) — then the next `run` / `op` sees the new value.

SEE ALSO

`altermod`, `foreach`, `@dev[param]`, `.param`.

SYNOPSIS

```
altermod modelname param = value
```

DESCRIPTION

Like `alter`, but targets a `.model` parameter — the change affects every device using that model. Useful for corner/temperature what-ifs.

```
└─ altermod.cir — run: ngspice -b altermod.cir
```

```
.model DMOD D(IS=1e-14)
V1 a 0 DC 0.7
R1 a d 100
D1 d 0 DMOD
.tran 1u 5u
.control
  run
  print v(d)[0]
  altermod DMOD IS=1e-11
  run
  print v(d)[0]          $ larger IS -> smaller forward drop
.endc
.end
```

```
→ output
v(d)[0] = 6.413270e-01
v(d)[0] = 4.950148e-01
```

Raising the diode saturation current by $1000\times$ lowers its forward drop from 0.641 V to 0.495 V (≈ 60 mV/decade $\times 3$ decades) ✓.

SEE ALSO

`alter`.

SYNOPSIS

```
@devicename[parameter]          $ e.g. @R1[resistance], @M1[gm], @Q1[ic]
```

DESCRIPTION

The `@` accessor reads a live device (or model) parameter or operating-point quantity into an expression — resistance of a resistor, `gm` / `vds` of a MOSFET, `ic` of a BJT, etc. It must be **evaluated** (via `let` / `print`), not `echo` ed. Which parameters exist for a device is listed by `devhelp` / `showmod`.

```
└─ vectors.cir (same deck)
```

```
print @R1[resistance]    $ 1000
```

```
→ output
@r1[resistance] = 1.000000e+03
```

SEE ALSO

`let`, `devhelp`, `alter`.

SYNOPSIS

```
compose name values v1 v2 ...
compose name start=a stop=b step=d      (or  lin=n / log=n)
```

DESCRIPTION

Creates a vector from an explicit list of values, or as a linear/log ramp. Handy for building a sweep axis or a lookup table inside `.control`.

`compose.cir` — run: `ngspice -b compose.cir`

```
R1 1 0 1k
.control
  compose p values 1 2 4 8
  print p
  compose ramp start=0 stop=1 step=0.25
  print ramp
.endc
.end
```

→ output

```
p:      1  2  4  8
ramp: 0  0.25 0.5 0.75 1
```

SEE ALSO

`let`.

`.csparam` — expose a parameter to `.control`

batch + ui

SYNOPSIS

```
.csparam name = { expression }
```

DESCRIPTION

Copies a netlist parameter into the `.control` namespace (as a vector), where plain `.param` values are otherwise invisible. The bridge between parse-time parameters and the control script.

`csparam.cir` — run: `ngspice -b csparam.cir`

```
.param vsup=3.3
.csparam vsupply={vsup}
.control
  echo "supply is" $&vsupply      $ 3.3
.endc
.end
```

→ output

```
supply is 3.3
```

Gotcha — `.param` is not visible in `.control`. Inside a control block, `$vsup` / `&$vsup` do not see a netlist `.param vsup`. Use `.csparam` to copy it across (or `set` / `let` a fresh value in the control block).

SEE ALSO

`.param`, `set`, `let`.

`.dc` — DC sweep

batch + ui

SYNOPSIS

```
.dc srcname start stop step [ srcname2 start2 stop2 step2 ]
```

DESCRIPTION

Sweeps a source (voltage or current) — or a nested pair of sources — and records the DC solution at each step, producing curves vs. the swept value. The classic use is a device I-V characteristic.

PARAMETERS

parameter	meaning
srcname	independent source to sweep (e.g. <code>Vs</code>)
start stop step	sweep range and increment
second source	optional outer sweep → a family of curves

EXAMPLE
— diode I-V

❏ `dc.cir` — run: `ngspice -b dc.cir`

```
* Diode I-V curve via .dc source sweep
Vs a 0 DC 0
R1 a d 100
D1 d 0 DMOD
.model DMOD D(IS=1e-14 N=1)
.dc Vs 0 1 0.05
.control
  run
  meas dc vd_1v FIND v(d) AT=1.0
  meas dc id_1v FIND i(vs) AT=1.0
.endc
.end
```

→ output

vd_1v	=	6.84819e-01
id_1v	=	-3.15181e-03

At $V_s = 1\text{ V}$ the diode drops 0.685 V , so $I = (1 - 0.685)/100\ \Omega = 3.15\text{ mA}$ ✓.

EXAMPLE 2
— nested sweep (a family of curves)

❏ `dc_nested.cir` — run: `ngspice -b dc_nested.cir`

```
.model NM NMOS(level=1 kp=200u vto=0.5)
Vd d 0 DC 1
Vg g 0 DC 0
M1 d g 0 0 NM w=10u l=1u
.dc Vg 0 2 0.5 Vd 1 2 1      $ inner Vg sweep, outer Vd -> a family
.control
  run
  let id = i(vd)
  print id[3]                $ Vgs=1.5 on the first Vds curve
.endc
.end
```

→ output

id[3]	=	-1.00000e-03
-------	---	--------------

The outer `Vd` sweep repeats the inner `Vg` sweep for each drain voltage — the MOSFET I_D - V_{GS} family. At $V_{GS}=1.5\text{ V}$ (saturation) $I_D = \frac{1}{2} \cdot k_p \cdot (W/L) \cdot (V_{GS} - V_{to})^2 = 1.0\text{ mA}$ ✓ (`i(vd)` negative by source convention).

Gotcha — source-current sign. `i(vs)` is negative here (-3.15 mA): a voltage source’s branch current is measured flowing *into* its + terminal, so current *delivered* to the circuit reads negative. Negate it (`-i(vs)`) for the physical current — but note `meas ... FIND -i(vs)` fails to parse; measure `i(vs)` and negate the result, or sense the current through a series element.

SEE ALSO
`.op` , `.tran` .

SYNOPSIS

```
define name(args) expression
```

DESCRIPTION

Defines a reusable function of one or more arguments, usable anywhere an expression is (in `let`, `print`, plotting). Handy for repeated post-processing (dB, RMS, a custom figure of merit).

▸ `vectors.cir` (same deck) — the `define` lines

```
define sq(y) y*y
print sq(4)           $ 16
```

→ output
`sq(4) = 1.600000e+01`

SEE ALSO

`let`.

devhelp / devices / show / showmod — device & model introspection

batch + ui

SYNOPSIS

```
devhelp [ devtype [ param ] ]      $ list a device type's parameters
devices [ regexp ... ]             $ list device instance NAMES only
show   device|regexp ... [ : param ... ] $ instance operating-point values
showmod model|regexp ... [ : param ... ] $ model parameter values
```

DESCRIPTION

`devhelp` lists what parameters a device type accepts and reports; `show` / `showmod` print the live values for instances/models after a run — the readable companion to the `@dev[param]` accessor.

TAMING THE OUTPUT — REGEX DEVICES, CHOSEN PARAMS

On a real design `show` dumps ~50 parameters per BSIM device × dozens of devices — pages of scroll. Two features cut that down, and they combine:

- **Select devices by regular expression:** `show r.*` = every resistor, `show m.*` = every MOSFET, `show m.x1.*` = every device in subcircuit X1. (Plain names, `all`, and the `type:subckt:dev` syntax still work; a name with a bare `.` stays literal, so `show m.x1.m7` is exact.)
- **Select parameters after a `:`:** everything past the colon names the parameters to print — `show m1 : vgs vds gm id` shows only those four. (An invalid name for that model prints `?????????`.)

Together: `show m.x1.* : id vgs vds gm` → those four values for every MOSFET in X1, one tidy table instead of pages.

DEVICES — JUST THE NAMES

When you only want to know *which* devices exist (not their values), `devices` lists the instance names, one per line — the equivalent of piping `show` through `grep`. No argument lists them all; each argument is a regex (grep-style):

```
devices          $ every device instance
devices m.*      $ every MOSFET: m2 m1 m.x1.m3 ...
devices m.x1.*   $ the devices inside subcircuit X1
devices ^r ^c    $ every resistor and capacitor
```

Handy to discover names before a `show ... : params` or an `@dev[param]` query. (*New command in this build.*)

`devhelp.cir` — run: `ngspice -b devhelp.cir`

```
.control
devhelp resistor
.endc
```

→ output

```
Resistor - Simple linear resistor
103 rsh      inout Sheet resistance
106 narrow   inout Narrowing of resistor
109 short    inout Shortening of resistor
...
```

EXAMPLE

— regex device selection + chosen parameters

`show_regex.cir` — run: `ngspice -b show_regex.cir`

```
.model NM NMOS(LEVEL=1 VT0=0.7 KP=120u)
V1 in 0 DC 1
R1 in a 1k
R2 a b 2k
M1 b in 0 0 NM W=10u L=1u
M2 a in 0 0 NM W=4u L=1u
.tran 1u 2u
.control
run
show r.*                $ every resistor (all default params)
show m.* : vgs vds gm id $ every MOSFET, only these four params
.endc
.end
```

→ output

device	r2	r1
resistance	2000	1000
device	m2	m1
vgs	1	1
vds	0.9244	0.8164
gm	0.000144	0.00036
id	2.16e-05	5.4e-05

On the two_balance chip, `show m.x1.* : id vgs vds gm` reads back all seven MOSFETs of instance X1 with four columns each. (*Regex device selection is an extension in this build; the `: param` selection is standard ngspice.*)

SEE ALSO

`@dev[param]` , `.model` , `display` .

SYNOPSIS

```
diff plot1 plot2 [ vec ... ]
```

DESCRIPTION

Compares two saved plots vector by vector and reports only the entries that differ by more than the tolerances (`set diff_abstol` / `diff_reltol` / `diff_vntol`). The quick way to see what a parameter change actually moved.

EXAMPLE

`i diff.cir` — run: `ngspice -b diff.cir`

```
V1 in 0 DC 1
R1 in out 1k
R2 out 0 1k
.op
.control
  run                $ -> plot op1 : v(out) = 0.5
  alter r2 2k
  run                $ -> plot op2 : v(out) = 0.667
  diff op1 op2
.endc
.end
```

→ output
op1.out[0] = 5.000000e-01 op2.out[0] = 6.666667e-01

Each `run` stores a plot (`op1` , `op2`); `diff` then prints exactly the node that changed when R2 doubled — the divider output moved from 0.5 to 0.667 ✓.

SEE ALSO

`setplot` , `alter` , `display` .

SYNOPSIS

```
display [ name | keyword | regexp ... ]
```

DESCRIPTION

Shows the **status** (not the values) of the vectors in the current plot: each line is `name : type, real/complex, length` , with the scale tagged `[default scale]` . The quickest way to see what a run produced or what an analysis created (e.g. the sensitivity vectors of `.sens`). For the numbers use `print` ; for a waveform, `plot` . It reports the **current plot only** — `setplot` first to inspect another.

SELECTING WHAT TO LIST

argument	lists
<i>(none)</i>	every vector in the current plot (alphabetical)
<code>v(out) i(v1) ...</code>	just the named vectors
<code>all · allv · alli · ally · alle</code>	all vectors / voltages / currents / other / events
<code><regexp></code>	every name matching a regular expression (see below)

The variable `set nosort` lists in internal (creation) order instead of alphabetical.

EXAMPLE

— list by type, and by regex

`display.cir` — run: `ngspice -b display.cir`

```
V1 in 0 SIN(0 1 1k)
Vdd vdd 0 DC 5
R1 in out 1k
C1 out 0 100n
Rl out vdd 2k
.tran 20u 1m
.control
  run
  display allv          $ voltages only
  display .*branch      $ regex: every branch current
.endc
.end
```

→ output

in	: voltage, real, 59 long	<- display allv
out	: voltage, real, 59 long	
vdd	: voltage, real, 59 long	
v1#branch	: current, real, 59 long	<- display .*branch
vdd#branch	: current, real, 59 long	

An argument that isn't an exact name (or a `v()` / `i()` form or an `all*` keyword) and contains regex metacharacters is treated as a **POSIX extended, case-insensitive, grep-style** regular expression matched against vector names: `display .*branch` → all branch currents, `display ^v` → everything starting with `v`, `display o.*t` → `out`. Plain names still match exactly. (*Regex selection is an extension in this build; stock ngspice-46 accepts only names and the `all*` keywords.*)

EXAMPLE

— a hierarchical circuit (where regex earns its keep)

`regex_hier.cir` — run: `ngspice -b regex_hier.cir`

```
* two instances X1, X2 of a stage -> internal nodes x1.n1, x2.n1, ...
Vin in 0 DC 3
X1 in mid stage
X2 mid out stage
Rl out 0 1k
.subckt stage a z
  R1 a n1 1k
  R2 n1 n2 1k
  R3 n2 z 1k
.ends
.op
.control
  run
  display x1.*          $ only X1's internal nodes
  print "x1.*"          $ their voltages (print needs the quotes)
  unlet x2.*            $ drop X2's internals
  display .*n1          $ x2.n1 is now gone
.endc
.end
```

→ output

x1.n1	: voltage, real, 1 long	<- display x1.*
x1.n2	: voltage, real, 1 long	
x1.n1 = 2.571429e+00		<- print "x1.*"
x1.n2 = 2.142857e+00		
x1.n1	: voltage, real, 1 long	<- display .*n1 (only x1.n1 left)

On a real design the subcircuit-internal nodes number in the dozens — the two-stage balanced amplifier in [user_examples/two_balance.cir](#) exposes `x1.a`, `x1.b`, `x1.vp`, `x1.vcm`, `x2...` and the per-device `@m.x1.m7[...]` currents. There, `display x1.*` lists just one instance's nodes and `print "x1.*"` dumps them — regex turns “what's inside X1?” into one line.

Note — two red herrings on hierarchical nodes. (1) A batch `.op` prints an operating-point *device dump* at the end of the run; that is the `.op` card's own output, **not** anything `print` / regex did. (2) `print "x1.*"` silently skips **zero-length** vectors — e.g. the empty `@r.x1.r1[i]` placeholders that `.option savecurrents` leaves in an AC plot — so you get the real node voltages instead of a “vector not available” warning. Use `^x1\.` to match only names *starting* with `x1`. (excludes the `@...` device-current vectors entirely).

SEE ALSO

`setplot`, `print`, `plot`, `let`.

.disto — small-signal distortion analysis

[batch + ui](#)

SYNOPSIS

```
.disto dec|oct|lin n fstart fstop [ f2overf1 ]
```

DESCRIPTION

Computes steady-state harmonic (and, with `f2overf1`, intermodulation) distortion from the device small-signal series. Stimulus comes from sources carrying a `DISTOF1` / `DISTOF2` spec. Results land in two plots: `disto1` (2nd-order) and `disto2` (3rd-order); the node vectors are the distortion products.

`disto.cir` — run: `ngspice -b disto.cir`

```
.model DMOD D(IS=1e-14)
V1 in 0 DC 0.6 AC 1 DISTOF1 1
R1 in a 1k
D1 a 0 DMOD
.disto dec 5 1k 100k
.control
  run
  setplot disto1
  let hd2 = mag(a)
  print hd2[0]
  setplot disto2
  let hd3 = mag(a)
  print hd3[0]
.endc
.end
```

→ output

hd2[0] = 1.044775e+00	(2nd-harmonic product at node a)
hd3[0] = 2.018926e+00	(3rd-harmonic product)

Note. Only devices that implement the distortion model (diodes, BJTs, MOSFETs) contribute; a purely linear circuit yields zero. Results are raw product magnitudes, not a normalized THD — for THD from a large-signal run use `.four`.

SEE ALSO

`.four`, `.ac`, `setplot`.

echo — print text

[ui only](#)

SYNOPSIS

```
echo [ stuff ... ]
```

DESCRIPTION

Prints its arguments, expanding shell variables (`$var`). It does *not* evaluate vectors or expressions — use `$&vec` for a vector's value, or `print` for expressions.

`echo.cir` — run: `ngspice -b echo.cir`

```
.control
  set name = world
  let x = 6*7
  echo "hello" $name      $ shell variable
  echo "x =" $&x          $ vector value
.endc
```

→ output

```
hello world
x = 42
```

SEE ALSO

`print`, `set`.

SYNOPSIS

```
fft vector ...
```

DESCRIPTION

After a `.tran` run, computes the FFT of each vector into a new spectrum plot (frequency resolution = $1/\text{timespan}$). Use `mag()` / `db()` on the complex result. A windowed, more controllable alternative to `.four`; window via `set specwindow=...`.

EXAMPLE

— two-tone spectrum

```
! fftex.cir — run: ngspice -b fftex.cir
```

```
B1 s 0 V = sin(2*3.14159*1k*time) + 0.5*sin(2*3.14159*3k*time)
R1 s 0 1k
.tran 5u 10m
.control
  run
  fft v(s)
  let m = mag(v(s))
  meas sp a1k MAX m FROM=0.9k TO=1.1k
  meas sp a3k MAX m FROM=2.9k TO=3.1k
.endc
.end
```

→ output

```
FFT: Frequency resolution: 99.957 Hz, output length: 1005
a1k = 9.95417e-01 at= 9.99570e+02
a3k = 4.93659e-01 at= 2.99871e+03
```

Both tones are recovered at 1.0 / 3.0 kHz with a 2.01 : 1 amplitude ratio — matching the 1 : 0.5 input ✓.

Gotcha — `meas` needs a real vector. `meas sp ... mag(v(s))` fails (“no such vector as `mag(v(s))`”): `meas` can’t take a function call inline. Materialize it first with `let m = mag(v(s))`, then measure `m`.

EXAMPLE

— `psd`: power spectral density

```
! psd.cir — run: ngspice -b psd.cir
```

```
B1 s 0 V = sin(2*3.14159*1k*time)
R1 s 0 1k
.tran 5u 10m
.control
  run
  psd 1 v(s)          $ 1 = averaging length; result overwrites v(s)
  let p = db(v(s))
  meas sp pk MAX p
.endc
.end
```

→ output

```
PSD: Frequency resolution: 100 Hz, output length: 1005
pk = -4.60992e+01 at= 1.00000e+03
```

`psd len vec...` is a close cousin of `fft` that returns the *power* spectral density (magnitude squared, optionally smoothed over `len` bins). The peak lands on the 1 kHz tone ✓. Use `fft` when you want the complex spectrum, `psd` when you want power.

SEE ALSO

`.four`, `spec`, `meas`.

SYNOPSIS

```
fopen  handle file [mode]      $ open; handle name in $handle-var
fread  handle var [len]       $ read a line into $var; status in $len-var
fclose handle
```

DESCRIPTION

Line-oriented file reading for `.control` scripts (parameter tables, external stimuli).

Batch limitation. On this build `fread` can block indefinitely under `ngspice -b` (no interactive input pump). Reliable interactively; for batch pipelines read the file with an external tool and pass values in via `.param` / `.csparam`.

foreach — iterate over a word list

SYNOPSIS

```
foreach var word1 word2 ...
... body (use $var) ...
end
```

DESCRIPTION

Runs the body once per word, with `var` set to each word in turn (words are literal text — values, node names, filenames). The most common way to repeat a run over a set of parameter values.

`foreach.cir` — run: `ngspice -b foreach.cir`

```
.control
  foreach r 1k 2k 4k
    echo "R = $r"
  end
.endc
.end
```

→ output

```
R = 1k
R = 2k
R = 4k
```

`foreach_set.cir` — iterate a `set` list variable; run: `ngspice -b foreach_set.cir`

```
.control
  set rvals = ( 1k 2k 4k )
  foreach r $rvals          $ walk the list held in a variable, not literal words
    echo "R = $r"
  end
.endc
.end
```

→ output

```
R = 1000
R = 2000
R = 4000
```

The word list can come from a `set` list variable (`$rvals`) rather than being typed inline — handy when the values are built elsewhere. Note the suffixes are evaluated to plain numbers (`1k` → `1000`) when stored in the variable.

SEE ALSO

`while`, `repeat`, `.dc`.

SYNOPSIS

```
.four freq output_variable ...
```

DESCRIPTION

After a `.tran` run, computes the DC component, fundamental (at `freq`), and harmonics 2–9 of each output, plus total harmonic distortion (THD). Operates on the last period of the transient waveform.

EXAMPLE

— harmonics of a half-wave rectifier

```
! four.cir — run: ngspice -b four.cir
```

```
* Half-wave rectifier -> .four shows strong harmonics
Vs in 0 SIN(0 1 1k)
R1 in a 50
D1 a out DMOD
Rout out 0 1k
.model DMOD D(IS=1e-14 N=1)
.tran 2u 20m
.four 1k v(out)
.end
```

→ output

Fourier analysis for v(out):

No. Harmonics: 10, THD: 87.7379 %, Gridsize: 200, ...

Harmonic	Frequency	Magnitude	Norm. Mag
0	0	0.0703844	0
1	1000	0.128272	1
2	2000	0.0956183	0.745433
3	3000	0.0548493	0.427601
4	4000	0.0192777	0.150288

Half-wave rectification is strongly nonlinear → THD $\approx 88\%$, with a large 2nd harmonic (74 % of the fundamental) — exactly the even-harmonic-rich spectrum expected.

Gotcha — sampling & period matter. `.four` resamples the last period onto a fixed grid; too coarse a `.tran tstep`, or a `tstep` that isn't an integer number of periods of `freq`, inflates the reported THD with interpolation error. Use a small `tstep` and a `tstop` that is a whole multiple of $1/\text{freq}$.

EXAMPLE

— `fourier`: the on-demand `.control` form

```
! four_ctrl.cir — run: ngspice -b four_ctrl.cir
```

```
Vs in 0 SIN(0 1 1k)
R1 in a 50
D1 a out DMOD
Rout out 0 1k
.model DMOD D(IS=1e-14 N=1)
.tran 2u 20m
.control
  run
  fourier 1k v(out)      $ same maths as .four, run when you choose
.endc
.end
```

→ output

No. Harmonics: 10, THD: 87.7379 %, Gridsize: 200, ...

The bare command `fourier freq vec...` is the interactive twin of the `.four` card: identical result, but computed on demand inside `.control` (e.g. after an `alter`, or on a vector you just built with `let`) rather than automatically at end of run.

SEE ALSO

`.tran`, `fft`, `spec`.

SYNOPSIS

```
.global node ...
```

DESCRIPTION

Declares nodes that are the *same net everywhere* — reachable inside any subcircuit without being wired through its ports. The usual candidates are supplies and the substrate (`vdd` , `gnd` , `vss`). Ground node `0` is always global.

`global.cir` — run: `ngspice -b global.cir`

```
.global vdd
Vdd vdd 0 DC 5
Vin in 0 DC 1
X1 in out amp
.subckt amp i o
  R1 i o 1k
  R2 o vdd 3k      $ vdd here is the global node, not a port
.ends
.tran 1u 5u
.control
  run
  print v(out)[0]   $ 2.0
.endc
.end
```

→ output
v(out)[0] = 2.000000e+00

The subckt reaches `vdd` without it being a port; v(out) settles where the two dividers balance: $(1/1k + 5/3k)/(1/1k + 1/3k) = 2.0\text{ V}$ ✓.

SEE ALSO

`.subckt` .

goto / label — control-flow jump

SYNOPSIS

```
label name      goto name
```

DESCRIPTION

An unstructured jump within a `.control` block. Prefer the structured loops (`foreach` / `while` / `repeat`); `goto` is for the occasional escape or a small state machine.

`goto.cir` — run: `ngspice -b goto.cir`

```
.control
  let i = 0
  label top
  echo "i =" $i
  let i = i + 1
  if i < 3
    goto top
  end
.endc
```

→ output
i = 0
i = 1
i = 2

SEE ALSO

`while` , `foreach` .

SYNOPSIS

```
.ic      v(node) = value ...      $ forced initial value (with .tran uic)
.nodeset v(node) = value ...      $ hint for the DC operating-point solver
```

DESCRIPTION

`.ic` forces a node to a value at the start of a transient run **when the analysis uses `uic`**. `.nodeset` only *seeds* the DC solver (to pick between multiple solutions or aid convergence) and is released once DC converges — it does not force a transient start value.

EXAMPLE

— pre-charged capacitor decays

`ic.cir` — run: `ngspice -b ic.cir`

```
V1 in 0 DC 0
R1 in out 1k
C1 out 0 1u
.ic v(out)=1
.tran 10u 5m uic
.control
  run
  print v(out)[0]          $ 1.0 (the initial condition)
  meas tran vend FIND v(out) AT=5m $ decays ~ e^-5
.endc
.end
```

→ output
v(out)[0] = 9.999000e-01
vend = 6.73767e-03

C starts at 1 V and discharges through R ($\tau=RC=1$ ms); at 5 ms = 5τ it is $1\cdot e^{-5} = 6.7$ mV ✓.

Gotcha — `.ic` needs `uic`. Without `uic` on the `.tran` line, ngspice computes a normal DC operating point and the `.ic` values are ignored (they only bias the initial guess). `.nodeset` never forces a transient start.

SEE ALSO

`.tran (uic)`, `.op`.

SYNOPSIS

```
.if(expr)
... cards ...
.elseif(expr)
... cards ...
.else
... cards ...
.endif
```

DESCRIPTION

Includes netlist cards conditionally at **parse time**, based on a parameter expression — corners, options, or variant topologies selected from one deck.

▮ **ifelse.cir** — run: `ngspice -b ifelse.cir`

```
.param mode=2
.if (mode==1)
  R1 in 0 1k
.else
  R1 in 0 4k
.endif
V1 in 0 DC 1
.op
.control
  run
  print @R1[resistance]    $ 4k (the else branch, since mode=2)
.endc
.end
```

→ output
@r1[resistance] = 4.000000e+03

SEE ALSO

`.param`, `foreach`.

SYNOPSIS

```
.include filename
.lib filename section
```

DESCRIPTION

`.include` textually inserts another file (parameters, subcircuits, models). `.lib file section` pulls only a named section — the standard way PDKs ship process corners.

▮ **include.cir** + ▮ **params.inc** — run: `ngspice -b include.cir`

```
* params.inc: .param vval=5 rval=2k
.include params.inc
V1 in 0 DC {vval}
R1 in out {rval}
R2 out 0 {rval}
.tran 1u 5u
.control
  run
  print v(in)[0]    $ 5 (from the included vval)
  print v(out)[0]   $ 2.5
.endc
.end
```

→ output
v(in)[0] = 5.000000e+00
v(out)[0] = 2.500000e+00

SEE ALSO

`.param`, `.ic`, `.temp`. Related structural cards not yet detailed: `.global`, `.option`, `.csparam`, `.model`.

SYNOPSIS

```
let name = expression      $ create/assign a vector
print expression ...       $ evaluate and show
```

DESCRIPTION

let creates or overwrites a vector from an expression over existing vectors and constants; **print** evaluates any expression (arithmetic, node voltages **v(n)**, branch currents **i(vs)**, functions). Operators include **+** **-** ***** **/**, **%** (mod), comparisons, and many functions (**sqrt**, **abs**, **db**, ...).

▮ **vectors.cir** — run: **ngspice -b vectors.cir**

```
V1 in 0 DC 2
R1 in out 1k
R2 out 0 1k
.op
.control
  run
  let x = 2 + 3
  print x                $ 5
  print v(out)           $ 1  (2 V divided by two equal Rs)
  print @R1[resistance]  $ 1000 (device parameter)
.endc
.end
```

→ output
x = 5.000000e+00
v(out) = 1.000000e+00
@r1[resistance] = 1.000000e+03

SEE ALSO

set, **define**, **@dev[param]**.

linearize — resample onto a uniform time grid

SYNOPSIS

```
linearize [ vec ... ]
```

DESCRIPTION

A transient runs on an *adaptive* timestep, so its samples are unevenly spaced. **linearize** interpolates the vectors onto a uniform grid (step = the **.tran tstep**). Required before operations that assume even spacing — notably **fft** — and handy for exporting clean columns with **wrdata**.

▮ **linearize.cir** — run: **ngspice -b linearize.cir**

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 10u 1m
.control
  run
  let n1 = length(v(out))
  echo "raw (adaptive) points:" $&n1
  linearize
  let n2 = length(v(out))
  echo "linearized points:" $&n2
.endc
.end
```

→ output
raw (adaptive) points: 112
linearized points: 101

The adaptive run used 112 unevenly-spaced points; **linearize** resamples to 101 uniform points (1 ms / 10 μs + 1) ✓.

SEE ALSO

fft, **wrdata**.

listing — show the parsed circuit

ui only

SYNOPSIS

```
listing [ logical | physical | deck | expand | runnable ]
```

DESCRIPTION

Prints the current circuit as ngspice sees it after parsing — useful to confirm parameter substitution, subcircuit expansion (`expand`), or the exact runnable deck (`runnable`).

↳ `listing.cir` — run: `ngspice -b listing.cir`

```
V1 in 0 DC 1
R1 in out 1k
R2 out 0 1k
.op
.control
    listing
.endc
.end
```

→ output

```
3 : v1 in 0 dc 1
4 : r1 in out 1k
5 : r2 out 0 1k
```

SEE ALSO

`source` , `edit` .

load — read a raw file back

ui only

SYNOPSIS

```
load [ rawfile ... ]
```

DESCRIPTION

Reads a raw file (from `-r` or `write`) into a new plot, so an earlier run’s data can be re-examined, compared, or post-processed without re-simulating.

↳ `load.cir` — run: `ngspice -b load.cir`

```
.control
    run
    write /tmp/saved.raw v(out)
    destroy all
    load /tmp/saved.raw      $ data is back — no re-run
    print v(out)[0]
.endc
```

→ output

```
v(out)[0] = 0.000000e+00
```

SEE ALSO

`write` , `setplot` .

.meas / meas — extract figures of merit

batch + ui

SYNOPSIS

```
.meas analysis result FIND expr AT=t | WHEN expr=val
.meas analysis result MAX|MIN|AVG|RMS|PP|INTEG  expr [FROM=t TO=t]
.meas analysis result TRIG expr VAL=a RISE=n  TARG expr VAL=b RISE=n
.meas analysis result PARAM='expr'
```

Control-mode: same, without the dot (`meas tran ...`) — except `PARAM='expr'` , which is dot-card only (see the table below). `analysis` is `tran` , `ac` , `dc` , or `sp` .

DESCRIPTION

ngspice’s measurement command — the analog of SPICE `.measure` / Eldo `.EXTRACT` . It reduces an analysis result to a named scalar you can print, reuse in a later `.meas` , or feed to a script: peak/trough (`MAX` / `MIN`), average/RMS/peak-to-

peak/integral over a window, a value at a time or when a condition holds (`FIND ... AT / WHEN`), and delays/rise-times (`TRIG ... TARG`).

`.MEAS` VS. `MEAS` — SAME ENGINE, DIFFERENT ROLE

The dot-card and the control-mode command share one measurement engine: the grammar above (`TRIG/TARG` , `FIND/WHEN` , `MAX/MIN/PP/AVG/RMS/INTEG`) and the numbers are identical either way. They are **not** fully interchangeable, though:

	<code>.meas</code> (dot-card)	<code>meas</code> (in <code>.control</code>)
when it runs	automatically at the end of its analysis	on demand, after you call <code>run</code>
result reusable?	prints only — not exposed as a vector (<code>print name</code> afterward fails)	stores a vector you can compute with (<code>meas ... vpk MAX ... then let d = vpk*2</code>)
can measure derived vectors (from <code>let / fft</code>)	no — only the analysis output	yes (e.g. <code>let m=mag(v(s)) then meas sp ... MAX m</code>)
PARAM='expr' type	yes	no — rejected (“no such function as ‘param=’”); use <code>let</code>
<code>.options autostop</code> (end run early once met)	yes — shortens the transient	no — the run has already finished
needs <code>.control</code> ?	no — works in a plain batch deck	yes

Rule of thumb: if the number is the end product (a batch/regression figure of merit, or you want `autostop`) use `.meas` ; if the number is an *input to more work* — a `let` expression, a second measurement, or measuring a vector you built yourself — use the control-mode `meas` . The Q example below relies on exactly that reuse.

Chaining works only within one mode. A measurement can reference an earlier result *of the same kind*: `.meas` → `.meas` via the `PARAM=` type (`.meas tran vpk2 param='vpk*2'`), and control `meas` → `meas` because each result is a stored vector (`let scaled = vpk*2`). But a `.meas` result **cannot** be used by a control-mode `meas` — it is print-only and never enters the interactive namespace (`let / $var / meas ... WHEN =vpk` all fail with “*vpk not available*”). If you need a measured value inside a script, compute it with the control `meas` after `run` , not with `.meas` .

EXAMPLE
— series-RLC step (underdamped)

`meas.cir` — run: `ngspice -b meas.cir`

```
V1 in 0 PULSE(0 1 0 1n 1n 1 2)
R1 in a 20
L1 a out 1m
C1 out 0 1u
.tran 1u 1m
.control
  run
  meas tran vpeak MAX v(out)
  meas tran vfinal FIND v(out) AT=1m
  meas tran vavg AVG v(out) FROM=0.6m T0=1m
  meas tran vrms RMS v(out) FROM=0 T0=1m
  meas tran trise TRIG v(out) VAL=0.1 RISE=1 TARG v(out) VAL=0.9 RISE=1
.endc
.end
```

→ output

```
vpeak = 1.35098e+00 at= 1.04838e-04
vfinal = 1.00001e+00
vavg = 9.99799e-01 from=6.0e-04 to=1.0e-03
vrms = 9.97501e-01 from=0.0 to=1.0e-03
trise = 4.24622e-05 targ=5.74e-05 trig=1.50e-05
```

Damping $\zeta = (R/2) \cdot \sqrt{C/L} = 0.316 \rightarrow$ overshoot $e^{-\pi\zeta/\sqrt{1-\zeta^2}} \approx 0.35$, so the peak is 1.35 V ✓, reached at 105 μ s ($\approx$ half the ringing period), and the output settles to 1.0 V.

EXAMPLE
— extracting the
Q
of a resonator from its -3 dB bandwidth

`q_resonator.cir` — run: `ngspice -b q_resonator.cir`

```
* parallel RLC tank: Q = R*sqrt(C/L) ; here R=316.23, L=1u, C=1n -> Q=10
Iac in 0 AC 1
R1 in 0 316.23
L1 in 0 1u
C1 in 0 1n
.ac dec 2000 1e6 30e6
.control
  run
  meas ac zmax MAX vm(in)           $ peak impedance = R, at fo
  let thr = zmax/sqrt(2)           $ -3 dB level
  meas ac flo WHEN vm(in)=thr RISE=1 $ lower skirt
  meas ac fhi WHEN vm(in)=thr FALL=1  $ upper skirt
  let Q = sqrt(flo*fhi)/(fhi-flo)   $ Q = fo / bandwidth
  print Q
.endc
.end
```

→ output

```
zmax      = 3.16211e+02 at= 5.03567e+06
flo       = 4.78753e+06
fhi       = 5.29089e+06
q = 9.998790e+00
```

A current source drives the tank, so `vm(in)` is its impedance: it peaks at R at the resonance f_o , and the two frequencies where it falls to $R/\sqrt{2}$ bound the bandwidth. Combining `meas` (peak + both skirts) with `let` yields $Q = f_o/BW = 10$, matching $R\sqrt{C/L}$ ✓. The `RISE=1` / `FALL=1` qualifiers pick the lower and upper crossings.

SEE ALSO

`.tran`, `print`, `.four`.

`.model` — named device model

batch + ui

SYNOPSIS

```
.model name type ( param=value ... )
```

`type` is a device kind: `D`, `NPN` / `PNP`, `NMOS` / `PMOS`, `R`, `C`, `SW`, ...

DESCRIPTION

Defines a named parameter set shared by every instance that references it. The instance card names the model in its model position (e.g. `D1 a k MYDIODE`).

`model.cir` — run: `ngspice -b model.cir`

```
.model MYDIODE D(IS=1e-14 N=1.2 RS=5)
V1 a 0 DC 0.7
D1 a 0 MYDIODE
.tran 1u 5u
.control
  run
  print -i(v1)[0]      $ 61.7 uA
.endc
.end
```

→ output

```
-i(v1)[0] = 6.171025e-05
```

EXAMPLE

— an NPN transistor model, recovering β

`model_bjt.cir` — `run: ngspice -b model_bjt.cir`

```
* force 10uA base current, read back beta = Ic/Ib
.model MYNPN NPN(IS=1e-16 BF=100 VAF=80)
IB 0 b DC 10u
Q1 c b 0 MYNPN
VC c 0 DC 5
.op
.control
  run
  let beta = -i(vc)/10u      $ negate: current is into the VC source
  print beta
.endc
.end
```

→ output
beta = 1.052822e+02

β comes out just above the model's `BF=100` : the Early effect (`VAF=80`) lifts it to $BF \cdot (1 + V_{CE}/VAF) \approx 106$ at $V_{CE}=5\text{ V}$ ✓.

EXAMPLE
— an NMOS model, recovering the square-law current

`model_mos.cir` — `run: ngspice -b model_mos.cir`

```
* LEVEL-1 NMOS in saturation, drain current from KP/VT0
.model NMOS1 NMOS(LEVEL=1 VT0=0.7 KP=120u LAMBDA=0)
M1 d g 0 0 NMOS1 W=10u L=1u      $ drain gate source bulk model
VG g 0 DC 1.7
VD d 0 DC 3
.op
.control
  run
  let id = -i(vd)                $ negate: current is into VD
  print id
.endc
.end
```

→ output
id = 6.000000e-04

The `M` card names four nodes (drain, gate, source, bulk) then the model, with `W/L` per-instance. With $V_{GS}=1.7$, $V_{DS}=3$ (saturated) and `LAMBDA=0`, $I_D = \frac{1}{2} \cdot KP \cdot (W/L) \cdot (V_{GS} - V_{TO})^2 = \frac{1}{2} \cdot 120\text{ }\mu\text{A} \cdot 10 \cdot (1.0)^2 = \mathbf{600\text{ }\mu A}$ ✓.

SEE ALSO
[altermod](#) (change a model at runtime), [OSDI](#).

.noise — noise analysis

batch + ui

SYNOPSIS

```
.noise v(output [,ref]) input_source dec|oct|lin n fstart fstop [ pts_per_summary ]
```

DESCRIPTION

Computes output-referred and input-referred noise spectral density vs. frequency (vectors `onoise_spectrum`, `inoise_spectrum`), and their integrals over the band (scalars `onoise_total`, `inoise_total`, in V and V/√Hz-integrated → V rms).

EXAMPLE
— the kT/C result

❯ **noise.cir** — run: `ngspice -b noise.cir`

```
* Output noise of RC low-pass -> integrates to sqrt(kT/C)
V1 in 0 DC 0 AC 1
R1 in out 1k
C1 out 0 100n
.noise v(out) V1 dec 100 0.1 100meg
.control
    run
    print onoise_total
.endc
.end
```

→ output
onoise_total = 2.035612e-07

The resistor’s thermal noise, low-pass filtered by C , integrates to the classic $\sqrt{kT/C} = \sqrt{(1.38\text{e-}23 \cdot 300 / 100 \text{ nF})} = 2.03\text{e-}7 \text{ V rms}$ — independent of R ✓. A textbook result the tool reproduces to three digits.

EXAMPLE

— reading the spectral density directly

❯ **noise_spec.cir** — run: `ngspice -b noise_spec.cir`

```
* thermal floor of a 1k resistor = sqrt(4kTR) = 4.07 nV/rHz, flat
V1 in 0 DC 0 AC 1
R1 in out 1k
C1 out 0 1p
.noise v(out) V1 dec 10 1 1k
.control
    run
    setplot noise1          $ spectra live in noise1, totals in noise2
    print onoise_spectrum[0]
.endc
.end
```

→ output
onoise_spectrum[0] = 4.071372e-09

$\sqrt{4kTR} = \sqrt{(4 \cdot 1.38\text{e-}23 \cdot 300 \cdot 1 \text{ k}\Omega)} = 4.07 \text{ nV}/\sqrt{\text{Hz}}$ ✓.

Gotcha — two plots, and the density is already *per* $\sqrt{\text{Hz}}$. `.noise` makes **noise1** (the `*_spectrum` vectors vs. frequency) and **noise2** (the scalar `*_total`). After `run` the current plot is noise2, so `setplot noise1` first to reach a spectrum. And `onoise_spectrum` is stored in $\text{V}/\sqrt{\text{Hz}}$, **not** V^2/Hz — don’t take a second square root.

SEE ALSO

`.ac`

SYNOPSIS

.op

Control-mode: op .

DESCRIPTION

Solves the quiescent DC bias: all capacitors open, all inductors shorted, sources at their DC value. Prints every node voltage and source current, and (in interactive mode) makes the small-signal device operating-point parameters available. It is the starting point of .tran (unless uic) and .ac .

EXAMPLE

— resistive divider

op.cir — run: ngspice -b op.cir

```
* DC operating point of a resistive divider
V1 in 0 DC 1
R1 in mid 1k
R2 mid 0 1k
.op
.end
```

```
→ output
Node                Voltage
in                   1.000000e+00
mid                  5.000000e-01
```

$v(\text{mid}) = 1\text{ V} \cdot R2 / (R1 + R2) = 0.5\text{ V} \checkmark$.

SEE ALSO

.dc , .tf , .tran .

SYNOPSIS

.option name[=value] ...

DESCRIPTION

Sets tolerances, temperature, integration method, and output behavior. The most common:

option	default	effect
reltol	1e-3	relative convergence/LTE tolerance
abstol / vntol	1e-12 / 1e-6	absolute current / voltage tolerance
trtol / chgtol	7 / 1e-14	transient LTE / charge tolerance (see the accuracy note)
temp / tnom	27 / 27	analysis / nominal temperature (°C)
method	trap	integration: trap or gear
bypass	0	device-eval bypass — leave off for precision analog
savecurrents	off	store branch currents (readable as @dev[i])
numdgt	6	digits in printed output
minres	0 (off)	parasitic reduction: short resistors below this value by merging their two nodes (ground and source terminals are protected)
mincap	0 (off)	parasitic reduction: delete capacitors below this value
pstran	off	pseudo-transient DC continuation as last-resort OP method, tried after Newton, gmin stepping, source stepping and optran have failed
dptran	off	damped pseudo-transient continuation (pstran plus a clamp on the Newton voltage update per iteration)
pstrancp	1n	pstran: artificial capacitance from every node to ground (F)
pstrandtinit	1n	pstran: initial pseudo timestep (s)
pstrandtmax	auto	pstran: final pseudo timestep; default pstrancp/gmin, where the artificial stamp has decayed to gmin
pstranmaxsteps	200	pstran: maximum number of pseudo timesteps

❏ option.cir — run: ngspice -b option.cir

```
.options savecurrents
V1 in 0 DC 1
R1 in mid 1k
R2 mid 0 1k
.op
.control
  run
  print @R1[i]      $ (1 - 0.5)/1k = 0.5 mA
  print @R2[i]
.endc
.end
```

→ output

```
@r1[i] = 5.000000e-04
@r2[i] = 5.000000e-04
```

SEE ALSO

.temp , @dev[param] , accuracy note.

The remaining `cp_coms` commands, for completeness. Aliases point at their documented twin; developer/debug dumps are grouped at the end.

command	mode	what it does
<code>setcs</code>	ui	like <code>set</code> but preserves the case of the value
<code>unset</code>	ui	remove a variable set by <code>set</code>
<code>undefine</code>	ui	remove a user function made with <code>define</code>
<code>newhelp</code> / <code>oldhelp</code>	ui	alternate front-ends to <code>help</code> (same content)
<code>trace</code>	batch+ui	print a node’s value at every timepoint (breakpoint family; remove with <code>delete</code>)
<code>iplot</code>	batch+ui	plot nodes <i>incrementally</i> as the run progresses (interactive display)
<code>bltplot</code>	ui	plot via the BLT widget toolkit (Tcl builds)
<code>use</code>	ui	load a compiled device library (like <code>codemodel</code> / <code>osdi</code>)
<code>deftype</code>	ui	redefine vector/plot type names (cf. <code>settype</code>)
<code>reshape -kin: shift</code>	ui	shift <code>argv</code> or a list variable one place left (scripting)
<code>cirbyline</code>	ui	enter a netlist one line at a time (programmatic deck building)
<code>setcirc</code> / <code>remcirc</code> / <code>removecirc</code>	batch+ui	switch to / delete the current circuit when several decks are loaded
<code>remzerovec</code>	ui	drop zero-length vectors from the current plot
<code>inventory</code>	batch+ui	print a count of devices by type in the circuit
<code>wrs2p</code>	ui	write S-parameter data as a Touchstone <code>.s2p</code> file (needs <code>.sp</code> + <code>P</code> ports)
<code>aspice</code> / <code>rspice</code> / <code>jobs</code>	ui	run a SPICE job asynchronously / remotely, and report on such jobs
<code>optran</code>	batch+ui	configure the OP→TRAN ramp-up (the 6 optran flags)
<code>cdump</code>	ui	dump the parsed <code>.control</code> structures (debug)
<code>mdump</code> / <code>mrdump</code>	batch+ui	dump the solver matrix / right-hand-side to a file (debug)
<code>check_ifparm</code>	batch+ui	validate device <code>IFparm</code> descriptors (developer)

SEE ALSO
`set` , `stop` (breakpoints), `osdi` .

SYNOPSIS

```
.param name = value    [ name2 = expr ... ]
.func  name(args) = { expression }
```

DESCRIPTION

`.param` defines named constants (usable in any value via `{name}`); `.func` defines a parameter-time function. Both are resolved during parsing — they cannot see simulation results (for that, use `let` / `define` in `.control`).

`└─ structure.cir` — run: `ngspice -b structure.cir`

```
.param mult=3 rtop=2k
.func scaled(x)={x*mult}
.subckt divider a b rt=1k rb=1k
    R1 a mid {rt}
    R2 mid b {rb}
.ends
V1 in 0 DC {scaled(1)}
X1 in 0 divider rt={rtop} rb=1k
.tran 1u 5u
.control
    run
    print v(in)[0]          $ scaled(1) = 3
    print v(x1.mid)[0]      $ 3 * 1k/(2k+1k) = 1
.endc
.end
```

→ output

```
v(in)[0] = 3.000000e+00
v(x1.mid)[0] = 1.000000e+00
```

EXAMPLE 2

— random parameters for Monte-Carlo (`agauss`)

`└─ mc.cir` — run: `ngspice -b mc.cir`

```
.param rval=agauss(1k,100,3)  $ gaussian: nominal 1k, ±100 at 3σ
V1 in 0 DC 1
R1 in 0 {rval}
.op
.control
    setseed 7
    run
    print @R1[resistance]      $ one random draw
.endc
.end
```

→ output

```
@r1[resistance] = 1.009399e+03
```

`agauss` / `gauss` / `unif` draw random parameter values (fixed with `setseed` for reproducibility) — the basis of a Monte-Carlo loop with `foreach` + `alter`.

Parse-time vs run-time. `.param` / `.func` are evaluated once at parse; they cannot reference node voltages or run results. Conversely `let` / `define` (in `.control`) work on simulation vectors and are *not* visible to element cards. Mixing the two is a common source of “parameter not found”.

SEE ALSO

`.subckt`, `let`, `define`.

SYNOPSIS

```
.plot analysis expr ...      $ batch: ASCII line plot
plot expr ... [ vs xexpr ]   $ control mode: graphical window
plot "regexp" ...           $ plot every matching vector
```

REGEX SELECTION

A **double-quoted** argument is a regular expression matched over the current plot (like `display`): `plot ".*#branch"` plots all branch currents. The quotes are required because bare metacharacters are expression operators; unquoted `vs/limit` keywords and expressions are untouched. (Applies equally to `gnuplot` and `hardcopy`, which share the plot engine.)

The `plot` vs `.plot` trap. They are *not* the same command. `.plot` (dot-card) renders an **ASCII** plot to the console and works in plain batch. `plot` (no dot, control mode) opens a **graphical** window and needs a `display` — in headless batch it produces no usable output. For a terminal/log or PDF pipeline, use `.plot` or `wrdata +gnuplot`, not `plot`.

EXAMPLE

— `.plot` ASCII output

↳ `dotplot.cir` — run: `ngspice -b dotplot.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 20u 1m
.plot tran v(out)
```

→ output

Legend: + = v(out)

```
-----
2.000e-05 1.264e-02 .          +          .
4.000e-05 4.518e-02 .          .+         .
6.000e-05 9.294e-02 .          . +        .
8.000e-05 1.532e-01 .          .  +       .
1.000e-04 2.224e-01 .          .   +      .
1.200e-04 2.974e-01 .          .    +     .
```

RELATED — `ASCIIPLOT`, `HARDCOPY`, `GNUPLLOT`

Three siblings render the same vectors different ways: `asciiplot vec...` draws the text plot shown above straight into the log (identical to a `.plot` card); `gnuplot file vec...` hands off to gnuplot; and `hardcopy file vec...` writes a PostScript file — the only one that works cleanly under `-b` for automated figure generation.

↳ `hardcopy.cir` — run: `ngspice -b hardcopy.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 20u 1m
.control
  run
  set hcopypscolor=1      $ colour instead of monochrome
  hardcopy plot.ps v(out)
.endc
.end
```

→ output

plot.ps : %!PS-Adobe-3.0 EPSF-3.0 (a ~9 kB Encapsulated PostScript figure)

SEE ALSO

`.print`, `wrdata`, `gnuplot`.

SYNOPSIS

```
print expr ...              $ control mode: dump values to stdout
print "regexp" ...          $ control mode: values of all matching vectors
.print analysis expr ...     $ batch: a formatted table
```

REGEX SELECTION

Because arguments are expressions (where `*`, `(`, ... are operators), a regular expression must be **double-quoted**: `print ".*#branch"` prints every branch current, `print "v.*"` every v-node — matched over the current plot (same regex as `display`). Unquoted expressions like `v(out)` or `vdb(a)-vdb(b)` are unaffected.

DESCRIPTION

Control-mode `print` writes the value of vectors/expressions to the console (a scalar prints on one line; a vector prints all points). The dot-card `.print tran|ac|dc ...` produces a numbered table of the named vectors after the run — and is one of the output directives that make an analysis actually run in batch mode.

EXAMPLE

— `.print` table

`dotprint.cir` — run: `ngspice -b dotprint.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 100u 500u
.print tran v(in) v(out)
```

→ output

Index	time	v(in)	v(out)
0	0.000000e+00	0.000000e+00	0.000000e+00
1	5.000000e-08	3.141593e-04	1.570011e-07
2	5.420140e-08	3.405574e-04	1.713021e-07
...			

EXAMPLE

— quoted regex over a

HIERARCHICAL

circuit

`regex_hier.cir` — run: `ngspice -b regex_hier.cir`

```
* two instances X1, X2 of a stage -> internal nodes x1.n1, x2.n1, ...
Vin in 0 DC 3
X1 in mid stage
X2 mid out stage
RL out 0 1k
.subckt stage a z
  R1 a n1 1k
  R2 n1 n2 1k
  R3 n2 z 1k
.ends
.op
.control
  run
  print "x1.*"          $ every node inside instance X1 (quotes required)
.endc
.end
```

→ output

```
x1.n1 = 2.571429e+00
x1.n2 = 2.142857e+00
```

One line dumps every node inside `X1`. On a real design this is the fast way to inspect a subcircuit — the two-stage amplifier in `user_examples/two_balance.cir` (see `.subckt`) exposes `x1.a`, `x1.b`, `x1.vp`, `x1.vcm`, `x2...`, so `print "x1.*"` reads back that whole instance at once. Two gotchas — both covered in the `display` entry — apply here: the quotes are required (bare `*` / `(` are operators), and a batch `.op` also prints a separate operating-point device dump that is **not** from `print`. Empty `@dev[i]` vectors (AC `savecurrents`) are skipped automatically; use `"^x1\"` to match only nodes and exclude device-current vectors.

SEE ALSO

`.plot`, `wrdata`, `display`.

SYNOPSIS

```
.pz node1 node2 node3 node4 cur|vol pz|pol|zer
```

DESCRIPTION

Finds the poles and/or zeros of a small-signal transfer function about the DC operating point. `node1 node2` are the input port, `node3 node4` the output port; `vol` (voltage transfer) or `cur` (transimpedance); `pz` = both, `pol` = poles only, `zer` = zeros only. Results appear as complex vectors `pole(k)` , `zero(k)` (rad/s).

EXAMPLE

— the single RC pole

`pz.cir` — run: `ngspice -b pz.cir`

```
* Pole-zero of an RC low-pass: v(out)/v(in)
V1 in 0 AC 1
R1 in out 1k
C1 out 0 100n
.pz in 0 out 0 vol pz
.control
  run
  print pole(1)
.endc
.end
```

→ output
pole(1) = -1.00000e+04,0.000000e+00

A single real pole at -10^4 rad/s = $-1/RC$ (= -1591 Hz), with no finite zero ✓. Poles are in rad/s; divide by 2π for Hz.

SEE ALSO

`.ac` , `.tf` .

SYNOPSIS

```
repeat [ n ]      if condition      break / continue
... body ...      ... body ...      (exit / skip a loop pass)
end               else
                  ... body ...
                  end
```

`repeat n` runs the body `n` times (forever if `n` is omitted — use `break` to leave).

`repeat.cir` — run: `ngspice -b repeat.cir`

```
.control
  let n = 0
  repeat 4
    let n = n + 1
    if n % 2 = 0
      echo $&n "is even"
    else
      echo $&n "is odd"
    end
  end
.endc
.end
```

→ output
1 is odd
2 is even
3 is odd
4 is even

SEE ALSO

`while` , `foreach` .

SYNOPSIS

```
reset      $ re-setup the circuit to its just-parsed state
rehash     $ rebuild the executable-command hash
help [subject]  quit      bug      tutorial
```

DESCRIPTION

`reset` restores the circuit to its initial state, undoing `alter` s and prior runs; `help` browses the built-in docs; `quit` exits; `bug` shows the bug-report address; `tutorial` points at the tutorials.

❯ `trivial.cir` — run: `ngspice -b trivial.cir`

```
V1 in 0 DC 1
R1 in 0 1k
.op
.control
  run
  print v(in)      $ 1.0
  reset
  run
  print v(in)      $ fresh setup, run again
.endc
.end
```

→ output
v(in) = 1.000000e+00
v(in) = 1.000000e+00

SEE ALSO

`alter`, `source`.

SYNOPSIS

```
reshape  vec ... [dim1,dim2,...]  $ change a vector's dimensions
transpose var ...                  $ transpose a multi-dimensional vector
cutout   vec ...                   $ keep only the current scale range
cross    newvec index vec ...      $ build a vector from one index across others
```

DESCRIPTION

Manipulate the shape/contents of data vectors — reshape a flat vector into a matrix, transpose it, trim to a window, or gather a cross-section. Mostly for post-processing multi-dimensional (e.g. swept) results.

SEE ALSO

`compose`, `let`.

DESCRIPTION

Without any `.save`, a `-r` run stores the **default set**: every node voltage and every voltage-source current. A `.save` (or control-mode `save`) replaces that with exactly the vectors you name, always alongside the `time`/frequency scale. Use it to shrink large raw files, or to store an expression such as a differential voltage.

REGEX SELECTION

`save` runs *before* the circuit, so a regex argument matches the **circuit node names** (not plot vectors): `save vd.*` saves every node starting with `vd`, `save .*out.*` every node containing `out`. The save spec forms `v(..) / i(..)`, `@dev[param]` and `all` are not treated as patterns. Bare regex is fine here (unlike `print`) because `save`’s arguments are names, not expressions.

EXAMPLE A

— default set vs. `.save`

```
└─ default.cir (no .save) — run: ngspice -b -r default.raw default.cir
```

→ output

```
No. Variables: 4
  0  time      time
  1  v(in)     voltage
  2  v(out)    voltage
  3  i(v1)     current
```

```
└─ restrict.cir — same circuit with .save v(out) — same command
```

```
* restrict the raw file
.save v(out)
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 10u 2m
.end
```

→ output

```
No. Variables: 2
  0  time      time
  1  v(out)    voltage
```

EXAMPLE 2

— store a computed expression (a differential voltage)

```
└─ save_diff.cir — run: ngspice -b -r out.raw save_diff.cir
```

```
V1 p 0 SIN(0 1 1k)
V2 n 0 SIN(0 0.7 1k)
R1 p 0 1k
R2 n 0 1k
.save v(p) v(n) vdiff=par('v(p)-v(n)')
.tran 50u 1m
.end
```

→ output

```
No. Variables: 4
  1  v(p)      voltage
  2  v(n)      voltage
  3  v(vdiff)  voltage ← the saved expression
```

`.save name=par('expr')` stores a computed vector straight into the raw file — the clean way to capture a differential output without post-processing.

EXAMPLE

— internal device parameters with `@device[param]`

```
└─ save_at.cir — run: ngspice -b save_at.cir
```

```
* @device[param] exposes a model's internal quantities
.model NMOS1 NMOS(LEVEL=1 VT0=0.7 KP=120u LAMBDA=0)
M1 d g 0 0 NMOS1 W=10u L=1u
VG g 0 DC 1.7
VD d 0 DC 3
.options savecurrents      $ makes @m1[id] @m1[is] ... available too
.op
.control
  run
  print @m1[gm] @m1[vdsat] @m1[id]
.endc
.end
```

→ output

```
@m1[gm]    = 1.200000e-03
@m1[vdsat] = 1.000000e+00
@m1[id]    = 6.000000e-04
```

Beyond node voltages and source currents, `@name[param]` reads a device's *internal* values — here $g_m = KP \cdot (W/L) \cdot (V_{GS} - V_{TO}) = 1.2 \text{ mA/V}$ and $V_{dsat} = 1.0 \text{ V}$ ✓. Hierarchical instances use the full path, e.g. `@m.x1.x2.m1[vgs]`. Per-terminal branch currents (`@m1[id]`, `@m1[is]`, ...) need `.options savecurrents`; you can `save` or `plot` any of these like an ordinary vector.

Gotcha — `.save` must not be the first line. The first line of a deck is always the **title** and is ignored. A deck that opens with `.save v(out)` silently saves *everything* because that line was treated as the title. (This exact mistake produced a wrong result while writing this entry — hence the leading comment line in the example above.)

.sens — sensitivity analysis

batch + ui

SYNOPSIS

```
.sens output_variable          $ DC sensitivity
.sens output_variable ac dec n fstart fstop $ AC sensitivity
```

DESCRIPTION

Computes the sensitivity of an output to every circuit element and model parameter, at the DC operating point (or over frequency with the `ac` form). Results land in a *Sensitivity Analysis* plot: the bare element name (e.g. `r1`) is $d(\text{output})/d(\text{element value})$, and `elem:param` (e.g. `r1:tc1`) breaks it down per model parameter.

EXAMPLE

— divider output sensitivities

```
! sens.cir — run: ngspice -b sens.cir
```

```
V1 in 0 DC 1
R1 in out 1k
R2 out 0 3k
.sens v(out)
.control
  run
  print r1 r2 v1      $ dV(out)/dR1, /dR2, /dV1
.endc
.end
```

```
→ output
r1 = -1.87500e-04
r2 =  6.249994e-05
v1 =  7.500000e-01
```

For $v(\text{out}) = V \cdot R2 / (R1 + R2) = 0.75 \text{ V}$: $dV/dR1 = -V \cdot R2 / (R1 + R2)^2 = -1.875\text{e-}4$, $dV/dR2 = +V \cdot R1 / (R1 + R2)^2 = +6.25\text{e-}5$, and $dV/dV1 = R2 / (R1 + R2) = 0.75$ — all matched ✓.

Note — batch needs an output directive. Like `.pz`, `.sens` only runs in `-b` with a `.control` block or a `.print` / `.plot` line (see §2). The sensitivity plot does not contain the output vector itself, so `print v(out)` alongside the sensitivities fails — print them in separate statements.

SEE ALSO

```
.tf, .op, .ac.
```

SYNOPSIS

set name = value	\$ define a shell variable
\$name	\$ expand a shell variable (text)
\$&vecname	\$ expand the numeric value of a vector

DESCRIPTION

`set` makes a **shell variable** (a string/flag, e.g. `set appendwrite`). `$name` substitutes its text. To drop a **vector’s numeric value** into a command line (e.g. an `echo` or a filename), use `$&vecname` — the `&` is what says “take the value of the vector”, not “look up a shell variable”.

▮ `vectors.cir` (same deck) — the `set / $` lines

```
set foo = hello
echo "foo is" $foo      $ shell variable -> text
let v = 7
echo "v ampersand:" $&v $ vector value -> text
```

→ output
foo is hello
v ampersand: 7

Gotcha — `$v` vs `$&v`. `$v` looks for a *shell variable* named `v`; if only the *vector* `v` exists, it expands to nothing (or errors). Use `$&v` for the vector’s value. This is the single most common `.control` scripting mistake.

SEE ALSO

`let`, `foreach` / `while`.

SYNOPSIS

```
setplot [ plotname ]      $ list plots / switch the current one
setscale [ vecname ]      $ set the default scale of the current plot
destroy [ plotname ... ]  $ free a plot's data (or 'all')
```

DESCRIPTION

Each run/analysis makes a new *plot* (a named vector set: `tran1`, `ac1`, `sens1`, ...) — so the plot list **is** your list of completed simulations. `setplot` with **no argument** lists them, marks the current, and shows each one's *analysis type* and title (and interactively prompts you to pick one); with a name it switches. Cross-plot references use `plot.vector` (e.g. `ac1.inp`). `destroy` reclaims memory.

LISTING PLOTS VS. VECTORS

command	lists
<code>setplot</code> (<i>no arg</i>)	the plots — one per analysis run — with type/title, current marked
<code>display</code> (<i>no arg</i>)	the vectors in the <i>current</i> plot
<code>echo \$plots</code>	plot names only, as a string — for scripting, not a listing

EXAMPLE

— several analyses, then list & switch

`! plotmgmt.cir` — run: `ngspice -b plotmgmt.cir`

```
V1 in 0 AC 1 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.control
  op          $ -> plot op1
  ac dec 10 1 1meg $ -> plot ac1
  tran 20u 1m  $ -> plot tran1
  setplot      $ list all plots, marks the current
  setplot ac1  $ switch to the AC plot
  display      $ vectors in ac1
.endc
.end
```

→ output

List of plots available:

```
Current tran1  * ... (Transient Analysis)
              ac1  * ... (AC Analysis)
              op1  * ... (Operating Point)
              const Constant values (constants)
```

Three runs → three plots, each tagged with its analysis. A second `tran` would add `tran2`, so the list doubles as a run history. `echo $plots` gives the same names tersely (`const op1 ac1 tran1`) when you need them in a script.

SEE ALSO

`display`, `reset`.

SYNOPSIS

```
setseed integer
```

DESCRIPTION

Fixes the seed of the random generator used by `agauss` / `gauss` / `unif` and Monte-Carlo draws, so a randomized run is exactly reproducible — set the same seed before each run to repeat a sequence.

SEE ALSO

`mc_source`, `.option`.

SYNOPSIS

```
settype type vec | regexp ...      $ type = voltage | current | time | frequency | impedance | ...
```

DESCRIPTION

Relabels a computed vector's physical type so plots and printouts carry the correct axis label and units (a derived impedance, power, gain, ...). A vector argument containing regex metacharacters retypes every current-plot vector whose name matches — e.g. `settype impedance .*#branch` (same regex selection as `display`).

```
❯ settype.cir — run: ngspice -b settype.cir
```

```
.control
run
let z = v(out)/1m
settype impedance z
display z
.endc
```

```
→ output
z : impedance, real, 59 long
```

SEE ALSO

`let`, `display`.

SYNOPSIS

```
shell [ command ]      cd [ dir ]      getcwd
```

DESCRIPTION

`shell` runs an OS command (or forks an interactive shell); `cd` / `getcwd` change and print the working directory. Handy for pre/post-processing around a run.

EXAMPLE

— backtick command substitution

```
❯ backtick.cir — run: ngspice -b backtick.cir
```

```
R1 1 0 1k
V1 1 0 DC 5
.op
.control
run
set nwords = `echo one two three four five | wc -w`      $ capture OS stdout
echo "the shell counted $nwords words"
let total = $nwords * 2                                   $ feed it into a let
print total
.endc
.end
```

```
→ output
the shell counted 5 words
total = 1.000000e+01
```

Backticks ``...`` run a command and paste its standard output straight into the `.control` line — the ngspice analogue of shell command substitution. Combined with `set` / `let` it lets an external tool (`sed` , `awk` , `wc` , a helper script) feed values back into the simulation.

Note. `shell` can block a non-interactive `-b` run if the command waits for input or a terminal; prefer running such steps outside ngspice in batch pipelines.

SYNOPSIS

```
snsave file      snload file
```

DESCRIPTION

Writes (or reloads) a full simulator snapshot — node values and internal state — so a long run can be checkpointed and continued later, or a hard-won operating point reused as a starting condition.

SEE ALSO

`.ic`.

SYNOPSIS

```
source file      edit [ file ]
```

DESCRIPTION

`source` reads and loads another netlist file into the running session; `edit` opens the current (or a named) deck in `$EDITOR` and reloads it on exit. Interactive workflow aids; use `.include` for static composition.

SEE ALSO

`listing`, `.include`.

SYNOPSIS

```
.sp  dec|lin n fstart fstop    $ S-parameters (needs P RF-port elements)
.pss fund tstop points ...    $ periodic steady state (shooting method)
.hb  fund nharm                $ harmonic balance
```

DESCRIPTION

`.sp` computes scattering parameters and requires `P` RF-port elements (each with a reference impedance) on the ports of interest. `.pss` (periodic steady state) and `.hb` (harmonic balance) are large-signal RF analyses for finding a circuit’s steady periodic response directly.

Build note. On this ngspice-46 build, `.sp` is present (but errors without RF ports: “*No RF Port is present*”), while `pss` and `hb` report “*no such command available*” — they need a build compiled with those experimental analyses enabled.

SEE ALSO

`.ac`, `.tf`.

SYNOPSIS

```
spec fstart fstop fstep vector ...
```

DESCRIPTION

Like `fft`, but computes the spectrum at the exact frequencies you name (`fstart` ... `fstop` by `fstep`), rather than the FFT's fixed 1/timespan bins. Run `linearize` first.

```
! spec.cir — run: ngspice -b spec.cir
```

```
B1 s 0 V = sin(2*3.14159*1k*time) + 0.5*sin(2*3.14159*3k*time)
R1 s 0 1k
.tran 5u 20m
.control
  run
  linearize
  spec 0 5k 100 v(s)
  let m = mag(v(s))
  meas sp a1k MAX m FROM=0.9k TO=1.1k
  meas sp a3k MAX m FROM=2.9k TO=3.1k
.endc
.end
```

→ output

```
a1k = 9.99901e-01 at= 1.00000e+03
a3k = 4.99552e-01 at= 3.00000e+03
```

Both tones land in exact 1.000 / 3.000 kHz bins with the right 2:1 amplitude ratio ✓.

SEE ALSO

`fft`, `.four`, `linearize`.

.step —

not supported

ngspice has no `.step`. A deck with `.step param ...` errors with “*unimplemented dot command '.step'*” — it is an LTspice/PSpice extension. To sweep a parameter across runs, use a control-mode loop with `alter` / `altermod`:

the ngspice way (no download — pattern)

```
.control
  foreach rl 1k 2k 3k
    alter R2 = $rl
    run
    meas tran vout FIND v(out) AT=...
  end
.endc
```

SEE ALSO

`foreach`, `alter`, `.dc` (built-in sweeps).

stop / resume / delete — breakpoints

batch + ui

SYNOPSIS

```
stop [ after n ] [ when expr ] ...    $ set a breakpoint / condition
resume                                $ continue a stopped run
delete [ all | number ... ]           $ remove breakpoints
```

DESCRIPTION

Pause a transient at a time, iteration, or condition to inspect state, then `resume`. Most useful interactively (or when driven from a script that examines vectors at each stop).

SEE ALSO

`.tran`.

SYNOPSIS

```
strcmp  var s1 s2          $ var = C strcmp(s1,s2): 0 if equal
strstr  var s1 s2          $ var = index of s2 in s1 (-1 if absent)
strslice var s1 offset len  $ var = s1[offset .. offset+len)
```

DESCRIPTION

Each sets a shell variable from a string operation, for scripting file names, labels, and conditionals in `.control`.

`strings.cir` — run: `ngspice -b strings.cir`

```
.control
  strcmp c "abc" "abc"
  strcmp d "abc" "abd"
  strstr p "hello world" "world"
  strslice s "abcdef" 2 3
  echo "equal=" $c "differ=" $d "pos=" $p "slice=" $s
.endc
```

→ output
equal= 0 differ= -1 pos= 6 slice= cde

SEE ALSO

`set`, `echo`.

.subckt / .ends / X — subcircuits

batch + ui

SYNOPSIS

```
.subckt name node1 node2 ... [ param=default ... ]
... elements ...
.ends
Xname net1 net2 ... name [ param=value ... ]      $ instantiation
```

DESCRIPTION

A subcircuit is a reusable block with external ports and optional parameters. Instantiate it with an `X` card, wiring your nets to its ports and overriding parameters by name. Internal nodes are addressed hierarchically as `Xinstance.node` (e.g. `v(x1.mid)` above).

EXAMPLE

— parameters: one block, three corner frequencies

`subckt_params.cir` — run: `ngspice -b subckt_params.cir`

```
* one parameterized RC low-pass, reused at three cutoffs
V1 in 0 AC 1
Xdef in def rclp          $ takes the default  c=1n  -> ~159 kHz
Xmid in mid rclp c=10n     $ override by name  -> ~15.9 kHz
Xlo  in lo  rclp c=100n    $ override by name  -> ~1.59 kHz
.subckt rclp a b r=1k c=1n  $ r,c are params with defaults
  R1 a b {r}               $ {...} evaluates the param
  C1 b 0 {c}
.ends
.ac dec 100 100 1meg
.control
  run
  meas ac fcdef WHEN vdb(def)=-3.0103
  meas ac fcmid WHEN vdb(mid)=-3.0103
  meas ac fclo  WHEN vdb(lo)=-3.0103
.endc
.end
```

→ output

fcdef	=	1.59155e+05
fcmid	=	1.59155e+04
fclo	=	1.59155e+03

Parameters are declared as `name=default` on the `.subckt` line and used inside as `{name}`. Each `X` card overrides by name (`Xmid ... c=10n`) or, like `Xdef`, inherits the default. The three cutoffs land exactly on $1/(2\pi \cdot 1 \text{ k}\Omega \cdot C)$ for $C = 1 \text{ n}, 10 \text{ n}, 100 \text{ n}$
✓ — one definition, three behaviours.

Gotcha — wrap parameter references in `{...}`. Writing `C1 b 0 c` makes ngspice parse the bare `c` as a *model* name and warn `can't find model 'c'`. The brace form `{c}` forces the token to be evaluated as the parameter. A literal number needs no braces; a parameter reference always does. (The `.subckt name ... params: c=1n` keyword before the defaults is an accepted synonym.)

EXAMPLE

— a two-stage balanced amplifier (two instances chained), extracting its differential input capacitance

`two_balance.cir` — full runnable set (models + stimulus) in `user_examples/`; run: `ngspice -b two_balance.cir`

```
* two instances of the same subcircuit, chained
xa vcase inp inn ib0 op   on   vdd gnd balance_simple
xb vcase op  on  ib1 outp outn vdd gnd balance_simple
.include ngspice_stimuli2          $ models + sources + .control

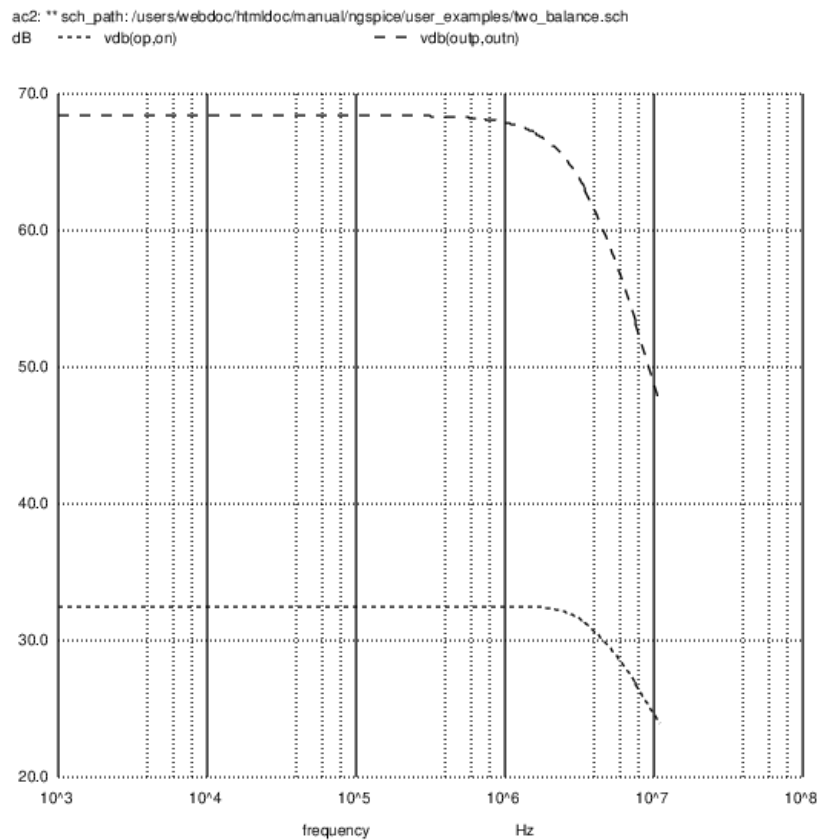
.subckt balance_simple vcase inp inn ib outn outp vdd gnd
M3 a inp vcm gnd nch w=10u l=0.18u m=1000    $ input diff pair
M1 b inn vcm gnd nch w=10u l=0.18u m=1000
M2 outn vcase a gnd nch w=10u l=0.18u m=100    $ cascode devices
M4 outp vcase b gnd nch w=10u l=0.18u m=100
M5 vcm ib gnd gnd nch w=10u l=0.6u m=100    $ tail current sink
M6 outn vp   vdd vdd pch w=5u l=0.25u m=500    $ PMOS load
M7 outp vp   vdd vdd pch w=5u l=0.25u m=500
R1 vp outn 10meg    $ R2 vp outp 10meg (common-mode feedback)
.ends
```

```
* --- ngspice_stimuli2 (excerpt): extract Cdiff from two AC runs ---
.control
ac dec 100 1k 11e6          $ ac1: common-mode drive
meas ac vinp_cm   find ac1.inp at=1e3
alter vinp acphase 0
alter vinn acphase 180      $ ac2: differential drive
ac dec 100 1k 11e6
meas ac vinp_diff find ac2.inp at=1e3
let cinp = 10p*(1/ac2.vinp_diff - 1/ac1.vinp_cm)/2
echo "Cdiff = ${cinp}"
.endc
```

→ output

```
Warning: command 'plot' is not available during batch simulation, ignored!
vinp_cm   = 4.887917e-01
vinp_diff = 4.803305e-01
Cdiff     = 1.80193E-13
```

The block is instantiated twice (`xa`, `xb`) and chained; the script drives it common-mode then differentially and computes the differential input capacitance, **Cdiff ≈ 0.18 pF**. Note the batch warning: the interactive `plot` is ignored under `-b` (use `.plot` / `gnuplot`) — see `$ plot` vs `.plot`. (*User-contributed example; generic LEVEL-3 models, no foundry data.*)



Passband differential gain ≈ 3.93 dB, flat to ~ 1 MHz, rolling off by ~ 10 MHz — the amplifier bandwidth. (This is what the batch-ignored `plot` would have shown; here it is drawn from the saved data.)

SEE ALSO

`.param`, `.include`, `.ac`, `meas`.

.temp — simulation temperature

[batch + ui](#)

SYNOPSIS

```
.temp value_in_celsius      $ or, in .control: set temp = value
```

DESCRIPTION

Sets the analysis temperature (°C). Every temperature-dependent model (diode/BJT/MOS saturation currents, resistor `tc1/tc2`, ...) is re-evaluated. In `.control` you can change `set temp` between runs to sweep temperature.

EXAMPLE

— diode current at 27 °C vs 100 °C

```
! temp.cir — run: ngspice -b temp.cir
```

```
.model DMOD D(IS=1e-14)
V1 a 0 DC 0.65
D1 a 0 DMOD
.tran 1u 5u
.control
  set temp = 27
  run
  print -i(v1)[0]      $ 0.82 mA
  set temp = 100
  run
  print -i(v1)[0]      $ 51 mA
.endc
.end
```

```
→ output
-i(v1)[0] = 8.204764e-04
-i(v1)[0] = 5.113547e-02
```

At a fixed 0.65 V the diode conducts 62× more at 100 °C — the strong temperature dependence of the saturation current ✓.

SEE ALSO

`altermod`, `.op`.

.tf — DC small-signal transfer function

[batch + ui](#)

SYNOPSIS

```
.tf output_variable input_source
```

DESCRIPTION

Computes, at the DC operating point, the small-signal ratio `output/input`, the resistance seen at the input source, and the resistance seen at the output. One command for gain and both port impedances.

EXAMPLE

— divider gain and impedances

```
! tf.cir — run: ngspice -b tf.cir
```

```
* DC small-signal transfer function
V1 in 0 DC 1
R1 in out 1k
R2 out 0 3k
.tf v(out) V1
.end
```

```
→ output
transfer_function      = 7.500000e-01
output_impedance_at_v(out) = 7.500000e+02
v1#input_impedance     = 4.000000e+03
```

Gain = $R2/(R1+R2) = 0.75$; $Z_{out} = R1 \parallel R2 = 750 \Omega$; $Z_{in} = R1+R2 = 4 \text{ k}\Omega$ ✓.

SEE ALSO

`.op`, `.ac`.

SYNOPSIS

```
.tran tstep tstop [ tstart [ tmax ] ] [ uic ]
```

As a control-mode command (inside `.control`), drop the dot: `tran tstep tstop ...`.

DESCRIPTION

Computes the circuit response in the time domain from `tstart` (default 0) to `tstop`. `tstep` is the printing/initial step; the actual internal step is chosen adaptively by local-truncation-error control and never exceeds `tmax` (default $(tstop-tstart)/50$). Results are stored in the current plot as vectors indexed by the `time` scale.

PARAMETERS

parameter	meaning
tstep	suggested/printing step size; also the initial step
tstop	end time
tstart	time at which output storage begins (analysis still starts at 0)
tmax	maximum internal timestep (caps the adaptive step)
uic	<i>use initial conditions</i> : skip the DC operating point; start from <code>.ic</code> / device <code>IC=</code> values instead

EXAMPLE

— RC low-pass step response ($\tau = R \cdot C = 1\text{ k}\Omega \cdot 1\text{ }\mu\text{F} = 1\text{ ms}$)

`rc.cir` — run: `ngspice -b rc.cir`

```
* RC low-pass step response
V1 in 0 PULSE(0 1 0 1n 1n 1 2)
R1 in out 1k
C1 out 0 1u
.tran 0.1m 5m
.control
  run
  meas tran v5ms FIND v(out) AT=5m
  meas tran trise TRIG v(out) VAL=0.1 RISE=1 TARG v(out) VAL=0.9 RISE=1
.endc
.end
```

`$ ngspice -b rc.cir`

→ output

No. of Data Rows : 79

v5ms = 9.93289e-01

trise = 2.19596e-03 targ= 2.30186e-03 trig= 1.05894e-04

The output settles to 0.993 V at 5 ms ($= 5\tau$, so $1 - e^{-5} \approx 0.993$ ✓), and the 10-90 % rise time is 2.196 ms $\approx 2.2\tau$, as expected for a single-pole RC.

EXAMPLE 2

— `uic`: an LC tank rings from an initial condition

`tran_uic.cir` — run: `ngspice -b tran_uic.cir`

```
L1 n 0 1m
C1 n 0 100n ic=1      $ start the cap at 1 V
.tran 1u 400u uic     $ no DC op; use the initial condition
.control
  run
  meas tran period TRIG v(n) VAL=0 FALL=1 TARG v(n) VAL=0 FALL=2
.endc
.end
```

→ output

period = 6.28842e-05

An LC tank has no DC operating point to settle to; `uic` starts it from `C1`'s `ic=1` and it oscillates at $T = 2\pi\sqrt{LC} = 2\pi\sqrt{(1\text{ mH}\cdot 100\text{ nF})} = 62.8\text{ }\mu\text{s}$ ✓.

Gotcha — `uic` skips the operating point. With `uic`, ngspice does *not* solve the DC bias first; every node not pinned by an `.ic` or a device `IC=` starts at 0 V. Use it for ring oscillators and switched circuits that have no valid DC solution, not as a convergence shortcut.

SEE ALSO

`.save` (restrict what is stored), `.ic` / `.nodeset`, control-mode `tran`, `meas`.

unlet — delete vectors

ui only

SYNOPSIS

```
unlet vecname | regexp ...
```

DESCRIPTION

Removes vectors from the current plot (frees memory, or drops intermediates before a `write`). The inverse of `let`. An argument containing regex metacharacters deletes every current-plot vector whose name matches — e.g. `unlet temp.*`, `unlet .*#branch` (same grep-style regex selection as `display`; the scale vector is protected).

SEE ALSO

`let`, `display`, `destroy`.

version / sysinfo / rusage / history — session info

batch + ui

SYNOPSIS

```
version [n]      sysinfo      rusage [resource]  history [-r] [n]
```

DESCRIPTION

`version` prints the ngspice version (and can assert a minimum); `sysinfo` reports CPU/memory; `rusage` prints resource counters (time, memory, iterations) — useful for benchmarking; `history` lists past commands.

`└ info.cir — run: ngspice -b info.cir`

→ output
** ngspice-46 : Circuit level simulation program [ngspice-jit fork: behavioral foundry R/C JIT-compiled to native OSDI]

SEE ALSO

`.option`.

SYNOPSIS

```
where      $ node/device with the largest residual (non-convergence)
dump       $ the circuit's node/branch matrix
state      $ current analysis state
status     $ analyses in progress and breakpoints
```

DESCRIPTION

Convergence and state introspection. `where` names the node or device carrying the largest residual after a failed Newton step — the first thing to check on a “no convergence” error; `dump` prints the raw matrix; `state` / `status` report the analysis and any breakpoints.

```
! where.cir — run: ngspice -b where.cir

→ output
No unconverged node found.
v(out) = 5.000000e-01
```

Title-line reminder. While writing this entry, a test deck put `V1 ...` on line 1 — the *title* — so the source vanished and the divider’s resistors were (correctly) removed as dangling, giving `v(out)=0`. Always start a deck with a comment/title (see §1).

SEE ALSO

`.option` (tolerances), `stop`.

SYNOPSIS

```
while condition      dowhile condition
... body ...         ... body ...
end                  end
```

`while` tests *before* the body (may run zero times); `dowhile` tests *after* (runs at least once).

```
! while.cir — run: ngspice -b while.cir

.control
let i = 0
while i < 3
echo "step" $&i
let i = i + 1
end
.endc
.end

→ output
step 0
step 1
step 2
```

`$&i` prints the numeric *value* of the vector `i` (see the Vectors chapter). A bare `$i` would look for a *shell variable* named `i`, which does not exist here.

SEE ALSO

`foreach`, `repeat`.

SYNOPSIS

```
wrdata file expr ...
```

DESCRIPTION

Writes the scale (time/frequency) and each expression as whitespace-separated columns of plain text — the easiest format to hand to gnuplot, a spreadsheet, or a script. Control mode only (no dot-card form). A **double-quoted** argument is a

regular expression over the current plot — `wrdata f.txt ".*#branch"` writes every branch current as columns (same rule as `print`).

EXAMPLE

`wrdata.cir` — run: `ngspice -b wrdata.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 50u 1m
.control
  run
  wrdata out.txt v(out)
.endc
.end
```

→ output

```
# out.txt : two columns, time then value
0.00000000e+00  0.00000000e+00
1.00000000e-07  6.27690799e-07
1.08405602e-07  6.84886513e-07
...
```

EXAMPLE

— several columns, one shared scale, with a header

`wrdata_multi.cir` — run: `ngspice -b wrdata_multi.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 100u 1m
.control
  run
  set wr_singlescale    $ one time column, not one per expression
  set wr_vecnames       $ prepend a header row of vector names
  wrdata multi.txt v(in) v(out)
.endc
.end
```

→ output

```
time          v(in)          v(out)
0.00000000e+00 0.00000000e+00 0.00000000e+00
1.00000000e-07 6.28318489e-04 6.27690799e-07
...
```

By default `wrdata` repeats the scale column *before every expression* (so N exprs give 2N columns). `set wr_singlescale` collapses that to one scale + N value columns; `set wr_vecnames` adds the name header — the two settings that make the file gnuplot- and spreadsheet-ready.

EXAMPLE

— `wrnodev`: snapshot node voltages as `.ic` lines

`wrnodev.cir` — run: `ngspice -b wrnodev.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 20u 1m
.control
  run
  wrnodev nodes.ic      $ present node voltages -> .ic cards
.endc
.end
```

→ output

```
# nodes.ic
* Recorded at simulation time: 0.001
.ic v(in) = -2.44929e-16
.ic v(out) = -0.450639
```

Unlike `wrdata` (which writes a whole waveform), `wrnodev` captures just the *present* operating state of every node as ready-to-`.include .ic` cards — a convenient way to seed another run from where this one stopped.

SEE ALSO

`print`, `write` (binary raw), `gnuplot`.

write vs. -r — two ways to make a raw file

ui only

DESCRIPTION

`-r file` dumps the run's default (or `.save`-restricted) set to *file*. `write file2 expr...`, inside `.control`, writes a chosen set to *file2*. They coexist only if the filenames differ. A **double-quoted** argument is a regular expression over the current plot — `write br.raw ".*#branch"` writes every branch current (same rule as `print`).

EXAMPLE B

— `write` to a *different* file than `-r`: both are produced

`both.cir` — run: `ngspice -b -r main.raw both.cir`

```
* write to a different file than -r
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 10u 2m
.control
  run
  write sel.raw v(out)
.endc
.end
```

→ output

```
main.raw : No. Variables: 4   (time, v(in), v(out), i(v1) — the -r default set)
sel.raw  : No. Variables: 2   (time, v(out) — exactly what write named)
```

EXAMPLE

— a human-readable (ASCII) raw file

`write_ascii.cir` — run: `ngspice -b write_ascii.cir`

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 100u 1m
.control
  run
  set filetype=ascii      $ default is binary
  write out_ascii.raw v(out)
.endc
.end
```

→ output

```
Title: * set filetype=ascii makes 'write' emit a human-readable rawfile.
Plotname: Transient Analysis
Flags: real
No. Variables: 2
No. Points: 63
Variables:
    0      time      time
    1      v(out)    voltage
Values:
0      0.0000000000000000e+00
      0.0000000000000000e+00
...
```

Rawfiles are binary by default (compact, exact). `set filetype=ascii` before `write` emits the same structure as text — readable, diff-able, and loadable back with `load`. It applies to `-r` too.

EXAMPLE

— `appendwrite`: many runs, one file

```
1 appendwrite.cir — run: ngspice -b appendwrite.cir
```

```
V1 in 0 SIN(0 1 1k)
R1 in out 1k
C1 out 0 100n
.tran 20u 1m
.control
  let rk = 1k
  while rk le 3k
    alter r1 rk
    run
    write sweep.raw v(out)
    set appendwrite      $ 2nd run onward: append, don't overwrite
    let rk = rk + 1k
  end
  destroy all           $ clear the in-memory run plots first
  load sweep.raw        $ ...so echo shows only what the FILE holds
  echo $plots
.endc
.end
```

→ output

```
const tran1 tran2 tran3
```

Normally each `write` overwrites; `set appendwrite` makes subsequent writes *append* a new plot instead — the idiom for sweeping a parameter with `alter` and keeping every run in one raw file. Reload it and each run reappears as `tran1`, `tran2`, ... (the `destroy all` is only so the echo isn't cluttered by the plots still live from this session).

Gotcha — `write` to the same file as `-r` loses. If `write` targets the very file passed to `-r`, the `-r` dump wins: the file ends up with the full default set (4 variables), and `write`'s selection is discarded. **Give `write` a different filename from `-r`** (or don't use `-r` at all).

Gotcha — bare `run` needs an analysis card. `run` executes a *previously defined* analysis (a `.tran` / `.ac` / ... card, or a `tran` issued earlier in `.control`). With no analysis defined, `run` does nothing and no raw file is written.

SEE ALSO

`.tran`, `wrdata` (ASCII columns), `print`, `load` (read a raw file back).

4. OSDI — compiling behavioral foundry passives into native devices

Foundry PDKs express precision passives as **behavioral equations** — a resistor or capacitor whose value is a function of terminal voltage and temperature (high-field `tanh`, `vc1/vc2` polynomials, temperature written into the expression). Stock SPICE can't model those directly, and the usual fallbacks hurt:

- A behavioral `C=f(v)` is realized as an auxiliary differentiator — numerically fragile, and on large designs it aborts with **“Timestep too small”**.
- A behavioral `R=f(v)` becomes a B-source `I = V/R(V)`; the foundry's `abs(V)` puts a slope discontinuity right at the operating point, hurting the Newton Jacobian.

This build adds a parse-time pass that instead **compiles the foundry's own equation into a native OSDI device** using the system `gcc` — no OpenVAF, no Verilog-A, no model edits. Resistors become a native current device with an analytic Jacobian; capacitors become a **charge-conserving** native charge device (so they are stable where the differentiator was not). Identical laws are compiled once and shared (structural dedup); `m` (mfactor) is first-class; and electrically inert caps (shorted `n1==n2`, or dangling on a floating node) are dropped, matching native topology reduction. It is **on by default**; `set osdi_res_bsource` forces the old behavioral path.

HOW IT ENGAGES

Automatic: any behavioral resistor (`R='...v(a,b)...`) or capacitor (`C='...v(a,b)...`) in the netlist is compiled at parse time. The run log reports what was generated, e.g. `osdi-res: N resistor(s) + M capacitor(s) -> K C-OSDI model(s)` .

TURNING IT OFF — SET `OSDI_RES_BSOURCE`

The codegen is the **default**; one boolean variable is the global opt-out. With `set osdi_res_bsource` , every behavioral `R=` / `C=` keeps its original **B-source** implementation instead — no compilation, no OSDI device. It governs resistors *and* capacitors together.

```
* in .spiceinit (it is read before the deck is parsed):
set osdi_res_bsource      $ keep all behavioral R/C as B-sources, skip OSDI codegen
```

Put it in `.spiceinit` , not in the deck's `.control` : codegen runs at **parse time**, before an in-deck `.control` block executes, so setting it there would be too late. Verified — with the flag the `osdi-res:` line disappears and the circuit runs the behavioral network. Two more behaviors: even with codegen on, a device whose expression uses an unsupported function, or a system with no `gcc` , **falls back to B-source automatically** (warned once); and `set osdi_res_include="/path"` overrides the C header include directory used when compiling the generated model.

EXAMPLE

— generic voltage-dependent R and C (no foundry data)

```
❯ osdi_selftest.cir — run: ngspice -b osdi_selftest.cir
```

```
.param r0=1000 k=0.1 c0=1e-12 vc=0.05
* voltage-dependent resistor  R(v) = r0*(1 + k*v^2)
vr a 0 dc 1
Rtest a 0 R='r0*(1 + k*v(a,0)*v(a,0))'
* voltage-dependent capacitor  C(v) = c0*(1 + vc*v)
vx n 0 dc 0 pulse(0 1 1n 1n 1n 20n 40n)
Rin n x 1k
Ctest x 0 c='c0*(1 + vc*v(x,0))'
.control
  op
  print i(vr)
  tran 0.5n 40n
.endc
.end
```

→ output
`osdi-res: 1 resistor(s) + 1 capacitor(s) -> 2 C-OSDI model(s); 0 kept behavioral; 0 shorted/dangling cap(s) dropped`
`i(vr) = -9.09091e-04`

The two behavioral expressions are compiled into two native OSDI devices, and the DC check is exact: at $v(a)=1$ V, $R = r_0(1 + k \cdot 1^2) = 1100 \, \Omega$, so $i(vr) = -1/1100 = -9.09e-4$ A ✓ (negative by source convention — see `.dc`).

Precision analog: leave `.option bypass` off. On very-high-gain sample-and-hold / ADC nodes, `.option bypass=1` lets a bounded ~ 1 mV error meander through the hold — a property of the *bypass approximation*, not of OSDI (native devices show it too). Bypass is off by default; keep it off for precision analog. Full analysis in [docs/osdi_bypass_wobble/](#) of the ngspice source tree.

MANUAL CODEGEN — `OSDIGEN` / `OSDIGENC`

Besides the automatic path, two `.control` commands emit a C-OSDI model on demand:

```
osdigen  modelname n1 n2 expr  $ resistor  R(v) from a behavioral expression
osdigenc modelname n1 n2 qexpr  $ capacitor  from a charge expression Q(v)
```

Use them to pre-generate and inspect a model (e.g. to check the emitted C, or build a library) rather than letting the parser compile netlist behaviorals silently. Same generator, same output devices.

SEE ALSO

the OSDI codegen docs in the source ([docs/osdi_codegen_overview/](#) , [osdi_res_codegen/](#) , [osdi_cap_codegen/](#)).

tested examples, the niche/alias/developer remainder in the Other commands table.