PIX37 Manual

Editor, then code. Offline copy of the book on pix37.party/docs.

Editor

PIX37 DEV is the editor. You draw, write, and hear the game here, then export a cartridge or a standalone build. It can also play a cartridge, and it can open PIX37.PARTY to download cartridges and use the catalog. PIX37 DEV EDUCATIONAL is the same editor, and it is free. It has no builds, no cartridge export, and no access to PIX37.PARTY. PIX37 Console is the player. It plays `.p37` cartridges, and you open PIX37.PARTY to download cartridges and browse the catalog.

Projects and cartridges

A `.pix37` file is the project: source tabs, sprite sheets, maps, sound, and data tables. A `.p37` file is the cartridge the player runs: tightly packed bytecode.

Saves and achievements sit in the author folder next to the app data. On Windows that looks like `C:\\Users\\Ada\\AppData\\Roaming\\PIX37\\Mira\\Pong.p37s` for a save, and `C:\\Users\\Ada\\AppData\\Roaming\\PIX37\\Mira\\Pong.ach` for achievements. On Linux the same files are `~/.local/share/PIX37/Mira/Pong.p37s` and `~/.local/share/PIX37/Mira/Pong.ach`. Ada is the account on the computer. Mira is the author name, and that name is the folder. Pong is the save slot, which defaults to the game title. `save` writes the `.p37s`. The first unlocked achievement writes the `.ach`. If the files are already there, the runner reads them when the cartridge starts.

Dev menu

Escape opens the dev menu. Escape while a game is running returns to the editor. Confirm with Enter or the A button (V by default). Back is the B button (C by default).

With no project loaded the menu is: CREATE PROJECT, LOAD PROJECT, BUILD, LOAD CARTRIDGE, CONNECT TO PIX37.PARTY, SETTINGS.

With a project loaded it is: SAVE PROJECT, LOAD PROJECT, CREATE PROJECT, ASSETS, BUILD, LOAD CARTRIDGE, CONNECT TO PIX37.PARTY, SETTINGS.

Ctrl+S saves the open project. Ctrl+R runs it.

Create a project

CREATE PROJECT asks for the cartridge before you start making the project.

- **VIDEO** — 1-BIT (2 colors), 2-BIT (4), 4-BIT (16), 8-BIT (256).
- **AUDIO** — sample depth of the synth: 16, 8, 4, 2, or 1 bit.
- **SCREEN** — presets include 64×64, 128×128, 160×90, 256×256, and 320×180. CUSTOM size lets you pick any size from 1×1 to 512×512.
- **PALETTES** — banks of colors. The first bank starts as the 16-color set below. LOAD PALETTE applies a built-in set: PICO-8 16, PICO-8 32, Game Boy, 1-bit, CGA, Sweetie-16, and a longer list taken from Lospec (Resurrect 64, Apollo, Endesga 32 and 64, CC-29, SLSO8, Pear36, Oil 6, Duel, AAP-64, Lost Century, and others). You can also load a palette file, or load your own PNG and take the palette out of that image.



Color indexes used by `cls`, `pset`, `spr`, and the rest. You can repaint any index in the palette editor.

The five panels

The editor is GAME, PAINT, CODE, SOUND, and DATA, in that order. The map lives inside PAINT. SFX and songs live inside SOUND. Click a panel to work there. When CODE has focus, SOUND and DATA ignore letter keys so typing does not also play the piano.



> Full layout, written `1:1:4:1:1`. Each number is how much of the row that panel takes: GAME, PAINT, CODE, SOUND, DATA. A small screen sits centered in a 128×128 cell. A screen bigger than 128 px widens the GAME column.

SETTINGS → GLOBAL → LAYOUT cycles the arrangements:

Code	What you see
`1:1:4:1:1`	Everything. GAME and SOUND stacked, PAINT and DATA stacked, CODE on the right.
`1:1:4:1:1 2X`	The same arrangement. CODE is drawn at twice the density, so twice as much text fits.
`1:0:1:0:1`	GAME, DATA, CODE.
`1:0:1:0:1 2X`	The same arrangement, with the denser CODE.
`0:1:1:1:1`	No game preview. PAINT, CODE, SOUND, DATA.
`0:1:1:1:1 2X`	The same arrangement, with the denser CODE.
`0:1:4:1:2`	Studio. Wide CODE, a taller DATA strip, PAINT and SOUND.
`0:1:4:1:2 2X`	The same arrangement, with the denser CODE.
`0:0:4:0:4`	CODE and DATA only, both large.
`0:0:4:0:4 2X`	The same arrangement, with the denser CODE.
`0:1:4:1:0`	PAINT, SOUND, and a wide CODE.
`0:1:4:1:0 2X`	The same arrangement, with the denser CODE.
`0:4:0:0:0`	PAINT only, 256×256, one pixel of the sheet to one pixel of the canvas.

A `2X` row keeps the same panels. The CODE panel is drawn at twice the resolution and scaled back into the same cell, so the text, tabs, and gutter are half as tall and twice as much source fits on screen.

GAME

The preview of the running cart. Click the GAME panel, or press Ctrl+R, to run that preview. SETTINGS → GAME PANEL is where the panel is configured:

- **Preview** — idle color, continuous, or auto reload when you edit.
- **Play in a new window** — the game opens beside the editor.
- **Display** — preview as RGB888 or RGB565.
- **GAME 64** — a 64×64 screen is either centered 1:1 in the cell, or scaled 2× to fill it.
- **Shaders** — CRT and filters on the game, separate from the filters on the editor.

- **Color** — background of the panel.

PAINT

Sprite sheets and, on the map side of the same panel, tile maps. Sheets can be larger than the screen. Sprite preview sizes are 8, 16, 32, 64, and 128 (128 only when the sheet is at least 128 px). Scroll the mouse wheel over the sprite preview to zoom it. Brush sizes are 1, 2, 3, and 4.

Key	Tool
B	Pencil
M	Rectangle select
Q	Lasso
S	Stamp
G	Bucket
I	Eyedropper
V	Move
L / Shift+L	Line / curve
U / Shift+U	Hollow rectangle / hollow ellipse
F / Shift+F	Filled rectangle / filled ellipse
D	Dither
[ ]	Smaller / larger brush
Ctrl+Z / Ctrl+Y	Undo / redo. Ctrl+Shift+Z also redoes.
Ctrl+C / Ctrl+X / Ctrl+V	Copy, cut, paste
Ctrl+A	Select the canvas

The map uses the same idea, with fewer tools: B pencil, M rectangle select, S stamp, V move, and Ctrl+Z, Ctrl+Y, Ctrl+C, Ctrl+V. Hold Alt to zoom the map texture. Brush size on the map stays with the paint tools.

Each sprite has eight flag bits. Paint sets them on the sprite; `fget` and `fset` read them in the game.

SETTINGS → PAINT EDITOR is where that panel is configured:

- **Palette** — which bank is drawn, plus load and save.
- **Preview** — RGB888 or RGB565.

- **Brush cursor** — the pointer while you paint.
- **Pixel snap** — how the brush snaps to pixels.
- **Big brush** — a large brush snaps from its top-left, or from its whole footprint.
- **Onion skin** — a ghost of other frames, and which direction that ghost faces.
- **Map zoom** — true pixels, or a free zoom.
- **Colors** — the paint UI colors.

CODE

Tabs are separate files inside the project. Click a tab name to rename it; Enter confirms. Several tabs that each define `_init`, `_update`, or `_draw` are stitched together in tab order: the first tab runs first, the last tab runs last. Code that is not inside a hook becomes part of `_init`. A cart with no hooks at all is treated as one `_draw`.

Key	Action
Arrows	Move the caret. Shift extends the selection.
Ctrl+Left / Right	By word.
Home / End, Ctrl+Home / End	Start or end of the line, or of the tab.
Tab	Insert two spaces. In the completion list, accept the name.
Enter	New line, indented like the previous one. Accepts a completion.
Ctrl+C / X / V	Copy, cut, paste.
Ctrl+Z / Ctrl+Y	Undo / redo.
Ctrl+A	Select the tab.
Ctrl+D	Duplicate the line or the selection downward.
Ctrl+/	Comment or uncomment with `--`.
Ctrl+F	Find. F3 jumps to the next hit. Escape closes the bar.
Ctrl+G	Go to a line number.
Escape	Closes find, go-to, or the hint list first. A second Escape leaves the editor for the dev menu.

Hints pop up while you type a function name. Enter or Tab inserts the name, and a `(` if COMPLETE (is on in settings. SETTINGS → CODE EDITOR:

- **Search** — Ctrl+F in the active tab, or across every tab.

- **Text high** — crop glyphs to 5 px, or show the full glyph.
- **String size** — font size of string literals.
- **Comment font** — 3×5, or the Fusion font of the current UI language.
- **Complete (** — whether completion types the parenthesis.
- **Code hints** — on or off.
- **Hints language** — the menu language, or English.
- **Caret space** — extra offset on empty lines.
- **Gutter** — line numbers.
- **Color** — background, comments, tabs, numbers, operators, keywords, strings, engine functions, your functions, variables, brackets, caret, selection, error line, warning.

SOUND

The panel has two tabs. SFX is one pattern: a short run of notes. SOUND is a song: it places those patterns on eight channels. TIC is how many ticks one step waits. The clock is 120 ticks a second, and one step is a sixteenth note, so 15 ticks is 120 beats per minute. A higher TIC is a slower pattern. Two patterns stay in step when one TIC divides the other: 15 and 30 do, 15 and 8 do not.

The piano plays when a note is selected with the mouse. White keys are Z X C V B N M (C D E F G A B). Black keys are S D G H J (C# D# F# G# A#). Hover any value and type a number, or hover it and scroll the wheel. Pitch numbers in code run from 0 (C-0) to 63 (D#5).

On one pattern, LOOP and LEN are the same two numbers. OUT above 0 loops from IN through OUT, and the title reads LOOP. OUT at 0 and IN above 0 is LEN: that many notes play from the start, then the pattern ends, and the title reads LEN. Both at 0 plays every step.

How much of the synth you can edit depends on the bit depth chosen for the cartridge. SETTINGS → SFX EDITOR sets that depth, whether a note auto-advances (off, note chain, or field chain), whether Space restarts playback or toggles start and stop, and the editor colors.

16-bit

Each step has a note, a volume, a wave, and an effect. Volume is 0 to 7. 0 is silent.

Waves:

- **0 triangle**
- **1 tilted saw**
- **2 saw**
- **3 square**
- **4 pulse** — a narrow pulse
- **5 organ**
- **6 noise**

- **7 phaser**
- **8 PWM** — a square whose width moves
- **9 pulse 25%**
- **10 sine**
- **11 metallic noise**
- **12 FM bell**
- **13 pluck**
- **14 supersaw**
- **15 PWM (C64)**

Effects on a single note:

- **0 none**
- **1 slide** — pitch and volume glide to the next note across this step
- **2 vibrato** — pitch wobbles
- **3 drop** — pitch falls to silence, the usual kick
- **4 fade in**
- **5 fade out**
- **6 arpeggio** — walks this note and the next three, one every 8 ticks. At TIC 8 or below, every 4 ticks.
- **7 fast arpeggio** — the same four notes, every 4 ticks, or every 2 when TIC is 8 or below.

The pattern header, above the notes:

- **TIC** — ticks per step, from 1 to 255. 15 is 120 beats per minute. Type the number, or step it by one.
- **LOOP IN / OUT** — with OUT above 0, the pattern loops from IN to OUT. Leave OUT at 0 and the title becomes LEN: IN is how many notes play, and 0 means the whole pattern.
- **NOISE** — the whole pattern becomes noise.
- **CRUSH** — a buzzy variant of each wave.
- **DTUNE** — 0 off, 1 a light chorus, 2 a strong chorus.
- **RVERB** — reverb: 0 off, 1 short, 2 long.
- **DAMPN** — a low-pass filter: 0 off, 1 light, 2 strong.
- **LEVEL** — volume of the whole pattern, 0 to 7.
- **SWING** — shuffle. Odd steps wait a little longer, even steps a little less.
- **STEPS** — how many steps the pattern uses: 32, 24, 16, or 12.

8-bit

A step has a note and a wave. The four waves are square, triangle, saw, and noise. The header has TIC, LOOP IN / OUT, FX, and LEVEL. TIC is one of 1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 16, 20, 24, 30, 40, 60. FX is one effect for the whole pattern: 0 none, 1 vibrato, 2 arpeggio, 3 fade out. LEVEL is 0 to 3.

4-bit

A step is only a note. The header has TIC, LOOP (on or off), WAVE (the same four waves), and OCT (which octave the notes sit in). TIC is one of 4, 6, 8, 10, 12, 15, 20, 30.

2-bit

A step is empty, a drum, a hit, or crush. The header has TIC and BASE. TIC is 8, 15, 16, or 30. BASE sets how high the drum and the hit sit.

1-bit

A step is a note. TIC is 8, 15, 16, or 30, the same four speeds as 2-bit.

Songs

The SOUND tab is the open song. Eight columns are the channels. A cell is a pattern id. 255, drawn empty, means that channel is silent on that row. The X column is a command: continue, stop, loop start, or loop end. Stop ends the song after the row. Loop end jumps back to the last loop start.

A row ends when the left-most pattern with LOOP OUT at 0 finishes. Patterns further right do not make that row longer: a longer one is cut, a shorter one starts again at its first step until the row ends. A pattern with LOOP OUT above 0 loops between LOOP IN and LOOP OUT and does not end the row. If every pattern on the row loops, the row stays there.

Shortcuts

Key	Action
Piano keys	Write a note on the step selected with the mouse.
Up / Down	Change the value under the cursor.
Space	Play the open SFX, or the open song from the first row. Start/stop or restart, from SFX EDITOR settings.
Ctrl+Z / Ctrl+Y	Undo / redo.
Ctrl+C / Ctrl+X / Ctrl+V	Copy, cut, paste the selection.

DATA

Named tables. Each table is rows and columns. From code, `dget`, `dset`, `dstr`, `drows`, and `dcols` use these tables. Row and column numbers in code start at 0.

Column types, cycled on the header:

- **bool** — one bit.
- **2** — two bits.
- **4** — four bits.
- **8** — a byte.
- **16** — 16 bits, for ids and larger numbers.
- **spr** — a sprite id.
- **str** — text, drawn with the system font.
- **str8** and **str10** — the same text, drawn taller.
- **f16** and **f32** — a half-precision float and a full float.

With the pointer on a column header, Ctrl+scroll changes that column's type. A plain scroll moves across columns or down rows, whichever SETTINGS → DATA EDITOR selects. Ctrl swaps that direction, which is why Ctrl+scroll on a header changes the type when the wheel is set to columns.

Arrow keys move. Enter edits a cell. SETTINGS → DATA EDITOR:

- **Wheel** — scroll columns, or scroll rows.
- **Colors** — the grid, the header, and the selection.

Assets

ASSETS imports and exports the piece that is open, not the whole project. Sprites use the sprite open in PAINT, on the sheet that is open. The map is the map that is open. SFX is the pattern that is open. Music is the song that is open. Data is the table that is open. Code is the exception: every tab is written, and every tab is read back.

Kind	File	What moves
Sprites	`.png`	The open sprite.
Map	`.p37map`	The open map.
SFX	`.p37sfx`	The open pattern.
Music	`.p37mus`	The open song. Pattern numbers stay as they are. Export the SFX separately if you need those sounds.
Data	`.p37dat`	The open table.
Code	`.lua`	Every tab.

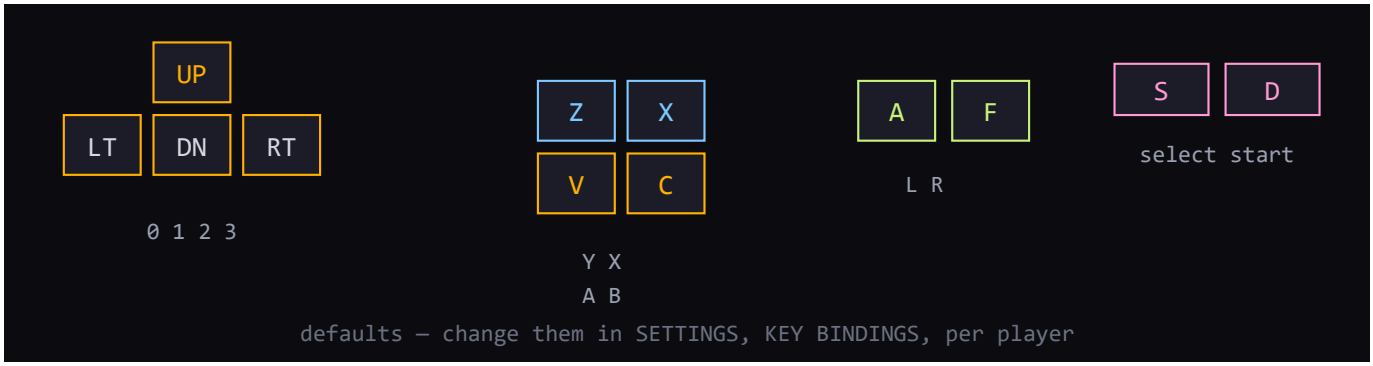
Settings

The root list:

Page	What it stores
PROFILES	Save and load every setting. New, rename, overwrite, delete. Copy or paste the profile as text.
Menu	Window, keys, menu colors, optional background.
GAME PANEL	Preview and the play window.
CODE EDITOR	Highlight and hints.
PAINT EDITOR	Brush, palette, map view.
SFX EDITOR	Synth and note entry.
DATA EDITOR	Wheel direction and colors.
LANGUAGE	Menu and hint language.
LICENCES	Licence texts, including the Fusion Pixel font and SDL3.

GLOBAL:

- **LAYOUT** — panel arrangement, above.
- **VOLUME** — master level, 0 to 10.
- **KEY BINDINGS** — keys for each of 8 players. Press a key to capture it. Directions, A B X Y L R, select.
- **Present** — skip idle frames, or draw every frame.
- **Fullscreen** — borderless.
- **Refresh** — window FPS. Left and right step 60, 90, 120, and max. Enter types a number up to 1000. 0 means no cap.
- **Vsync** — wait for the display between frames.
- **Scale** — integer window multiplier.
- **Shaders** — CRT and filters on the editor, not on the game.
- **Color** — menu text and highlight.
- **Background** — an optional PNG, cropped to 128, behind the menu. Clear removes it.
- **Editor cursor** — pointer shape.
- **Mouse** — floating pointer, or snapped to pixels.
- **UI speed** — how fast a held direction repeats through a list.
- **Explorer** — file lists start at the computer root, or at the folder of the exe.



Default keyboard. A is confirm in menus. B is back. In a game these are buttons 4 and 5. Select (S) is button 10, not a menu shortcut.

Button	Role	Default key
0	Left	Left arrow
1	Right	Right arrow
2	Up	Up arrow
3	Down	Down arrow
4	A, confirm	V
5	B, back	C
6	X	X
7	Y	Z
8	R	F
9	L	A
10	Select	S
11	Start	D

Build

Pick a platform: Windows, Linux, Mac, Android, iOS, or Web.

- **Name** — file name of the build.
- **Author** — on Windows and Linux. This is the folder name for saves.
- **Steam app id** — on Windows and Linux. Empty means no Steamworks. A filled id maps `achieve(0)`… `achieve(15)` to Steam achievements ACH_0…ACH_15.

- **Id** — bundle identifier where the platform asks for one.
- **Hide code** — on desktop and mobile overlays. It slows `peek` of the cart. It is not a lock.
- **Fullscreen** — the exported player starts fullscreen.
- **Shaders** — up to three layers baked into the player.
- **Download .p37** — Web only. Shows a download entry in the pause menu.
- **Icon** — a tile from a sprite sheet: sheet, size, x, y. Or a PNG cover you crop.
- **Export** — pick the folder. The menu reports BUILD SAVED or BUILD FAILED.

Make a build with the internet on. The splash may ask the server, anonymously, whether this build's hash is known. That check is there for two reasons. It protects me, the author of PIX37, from copies of the tool that were not mine to give away. It also protects the person who made the game. Taking someone else's game and listing it as your own is becoming common on the phone market, such as Play (Android) or iOS. The game still starts. If the machine is offline, the intro says the game is not connected. That line is also a hint to the player that the copy may not have come from a legal source. It is a small check, so that if your game was stolen and sold without your consent by someone else, you can make that case on a store such as Play (Android), Steam, or itch.io. The server stores the time that hash was first registered, next to the hash, and does not change it later. The earlier time is the original build. A hash registered after that is a later copy.

I want this check to protect the people who make games. The engine already does a lot of the work for them, and this is one of those things. I understand that it can be controversial. If the community decides it is better to drop this micro-DRM, I will.

An old cartridge from a previous editor can refuse to boot after the language changes. Open the `.pix37` project in this editor and export a new `.p37`.

Connect to PIX37.PARTY

CONNECT TO PIX37.PARTY is inside the app. There is no login page in the browser. You sign in with a Steam session ticket or with itch.io, using the pad and the mouse. The service shows a display name, 2 to 16 characters, unique. The licence key and the Steam id are not shown to other players.

After the terms of the service, the hub is Favorites, Catalog, Ranking, Players, and your profile.

A Developer licence is the editor. You make the game, you upload it, and you leave stars. A Gamer licence is the player. Hearts are from people playing on a runner.

- **Catalog** — browse games. Filters cover genre (up to three), screen size, cartridge size, and whether the upload includes a `.pix37` project.
- **Favorites** — games you marked.
- **Upload** — needs a Developer licence. You send a `.p37`, and you may attach the `.pix37` project. A `.p37` is packed bytecode. It is hard to read, and it can be obfuscated on purpose. A `.pix37` is the project file you actually work in: the source, the pictures, and the sound, in a form another person can open and learn from. Sending it lets other creators study the game instead of a compressed cartridge. Title,

author, genres, cover, icon, and achievement names go with it. The RAM number sent up is the peak the cart reached.

- **Comments and achievements** — on the game page. Unlocking an achievement while you are signed in is stored on that account as well as on the machine.
- **Ranking** — developers, sorted by stars or by hearts. Each row shows both counts. Open a name to see that author's games, ordered by hearts.
- **Players** — accounts ordered by achievements unlocked.

Code

The language is a small Lua-style script. This half starts from an empty project, teaches the language, and then shows a short example of every function.

A first program

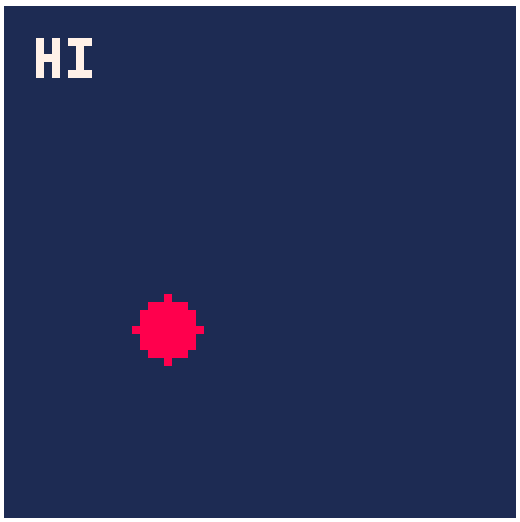
Create a project at 64×64, 4-bit color. Open CODE. Replace the tab with:

```
_init()
  x = 20
  y = 40
end

_update()
  if btn(0) then x -= 1 end
  if btn(1) then x += 1 end
  if btn(2) then y -= 1 end
  if btn(3) then y += 1 end
end

_draw()
  cls(1)
  print("HI", 4, 4, 7)
  circfill(x, y, 4, 8)
end
```

Ctrl+R runs it. Arrow keys move the circle. Escape stops the game and returns to the editor.



The same three calls: `cls(1)` paints the navy screen, `print` draws HI, and `circfill` draws the pink circle.

What runs, and when

- `_init` runs once.
- `_update` runs at 60 Hz. Put movement and buttons here.
- `_draw` paints. With no fps argument it paints as fast as the computer and the monitor can go. `_draw(30)` caps painting at 30 fps and leaves logic at 60.
- `_set_fps(n)` changes that cap later. 0 clears it.

Names are case-insensitive; the editor stores them in lowercase. `function` and `fun` are the same keyword.

Language

Programming basics

A program is a list of lines the cartridge runs. Each heading shows the shape of the line first, then a small example. A line that starts with `--` is a comment: a note for you. The game skips it.

Variables

A variable is a name that holds a value. You pick the name. Later lines can read it, and they can store a new value in the same name.

The shape is `name = value`. `local name = value` keeps that name inside the current block, so the rest of the cartridge cannot see it. Without `local`, the name is global and every tab can see it.

```
hp = 3            -- global: the whole cart can read and change hp
local coins = 0   -- local: only this block can see coins
hp = hp + 1       -- read 3, add 1, store 4 back into hp
```

`nil` means there is no value. `true` and `false` are yes and no. A name you have never set reads as `nil`.

```
bonus = nil            -- nothing stored yet
if bonus == nil then   -- condition: is bonus empty?
  bonus = 0            -- body: store a number
end
```

Tables

A table holds many values under one name. A list uses numbers. The first item is 1, not 0. A record uses names. A table can hold another table.

List shape: `{ value, value }`. Record shape: `{ name = value, name = value }`. A list of records is a table whose items are tables.

```
-- a list. [1] is the first item
scores = {10, 20, 30}
print(scores[1], 1, 1, 7)      -- print draws text. this draws 10

-- a record. player.x is the field named x
player = {x = 10, y = 20}
print(player.x, 1, 8, 7)       -- draws 10

-- a list of records. enemies[1] is the first record
enemies = { {x = 4, y = 8}, {x = 20, y = 12} }
print(enemies[1].x, 1, 16, 7)  -- draws 4
print(enemies[2].y, 1, 24, 7)  -- draws 12
```

Condition and body

A condition is a question with a yes or no answer, such as `hp > 0` or `btnd(4)`. A body is the group of lines that run when you enter a block. `if` and every loop use that same shape: a condition, then a body, then `end`. `repeat` closes with `until` instead of `end`.

If

`if` runs its body only when the condition is true. `elseif` tries another condition. `else` runs when every condition was false.

Shape: `if condition then`, then the body, then `end`.

```
hp = 2
if hp > 0 then                -- condition: is hp above zero?
  print("ok", 1, 1, 7)        -- body: print draws the word ok
elseif hp == 0 then           -- another condition
  print("empty", 1, 1, 7)
else
  print("no", 1, 1, 7)        -- body when every condition was false
end

if hp > 0 print(hp, 1, 32, 7)  -- one call, no then/end. draws 2
```

Loops

A loop repeats a body. `while` checks the condition first and keeps going while it stays true. `repeat` runs the body, then checks, so the body always runs at least once. `for` counts. The last number is included. Without a third number the step is 1, so if the start is already past the end the body does not run. Counting down needs a negative step.

```
n = 0
while n < 3 do         -- condition: keep going while n is under 3
  n += 1
  print(n)             -- 1, then 2, then 3: how many times the body has run
end

n = 0
repeat
  n += 1
  print(n)             -- body first, then the condition. prints 1, then 2, then 3
until n > 2

for i = 1, 3 do        -- i is 1, then 2, then 3
  print(i)
end

for i = 0, 4, 2 do     -- start, end, step of 2
  print(i)             -- 0, then 2, then 4
end

for i = 3, 1, -1 do    -- counting down needs the step
  print(i)             -- 3, then 2, then 1
end

for i = 1, 5 do
  if i == 3 then break end  -- leave the loop when i reaches 3
  print(i)                  -- 1, then 2
end
```

Walking a table

`all` visits each item of a list, from the first. `pairs` visits each name and the value stored under that name.

```
t = {1, 2, 3}                -- list: t[1] is 1, t[2] is 2, t[3] is 3
for v in all(t) do           -- v is 1, then 2, then 3
  print(v)                   -- print draws that number
end

player = {x = 10, y = 20}    -- record: a field x and a field y
for k, v in pairs(player) do
  print(k)                   -- k is the name, "x" then "y"
end
```

Functions

A function is a named recipe. The shape is `function name(arguments)`, then a body, then `end`. You call it by writing the name and parentheses. `return` sends a value back, and it can send more than one. The arguments are locals of that function.

```
function greet(who)       -- who is a local that holds what you pass in
  return "hi "..who       -- .. glues two texts. return sends the result back
end

print(greet("ada"))       -- draws hi ada
```

```
function pair()
  return 1, 2             -- two values
end

a, b = pair()             -- a becomes 1, b becomes 2
```

`local function` is a function that can see the locals around it. That is a closure. A table can hold one, and you call it with a dot.

```
function make()
  local n = 0
  local m = {}
  m.bump = function()     -- this function sees n
    n += 1
    return n
  end
  return m
end

counter = make()
print(counter.bump())     -- 1
print(counter.bump())     -- 2
```

Yes, no, and choosing

`not` flips a condition. `and` is true only when both sides are true. `or` is true when either side is true. `a and b or c` picks `b` when `a` is true, otherwise `c`. `cond ? a : b` is the same idea in one line: `a` when the condition is true, `b` when it is not.

```
hp = 2
hit = false
night = false

if hp > 0 and not hit then         -- both parts are true, so the body runs
  print("safe", 1, 1, 7)           -- draws safe
end

label = hp > 0 and "ok" or "out"   -- hp is above 0, so label is ok
print(label, 1, 8, 7)

shade = night ? 1 : 7              -- night is false, so shade is 7
print(shade, 1, 16, 7)
```

Math and text

`+ - * /` are add, subtract, multiply, and divide. `//` divides and keeps the whole number. `%` is the remainder. `^` is power. `..` glues two texts. `#t` is how many items are in a list. Comparisons are `==` (same), `~=` or `!=` (different), and `< > <= >=`. `x += 1` adds 1 to x. The same shape works for `-= *= /= ^= %=`.

```
scores = {10, 20, 30}    -- the list named scores
tiles = 15 // 8          -- 1, the fraction is dropped
left = 15 % 8            -- 7, what is left over
title = "hi ".."ada"     -- the text hi ada
n = #scores              -- how many items are in the list named scores
print(n, 1, 1, 7)        -- 3
print(title, 1, 8, 7)

score = 0
best = 10
if score ~= best then    -- condition: score is not the best
  score += 10            -- add 10 to score
end
print(score, 1, 16, 7)   -- 10
```

Where a local lives

`local` lasts until the `end` that closes its block. `if` / `end` is its own block. You may mention a local above the line that sets it, but until that line runs the value is `nil`.

```
if true then
  print(coins)       -- nil: the next line has not run yet
  local coins = 5
  print(coins)       -- 5
end
-- coins does not exist out here
```

Jumping

`::label::` marks a line. `goto label` jumps there, but only inside the same function.

```
x = 0
::again::
x += 1                       -- add 1
if x < 3 then goto again end  -- condition: jump back while x is under 3
```

Memory, in short

`peek` and `poke` look at live pixels through one 16-bit address. They do not keep a second copy of the cartridge. Addresses `0x5F00` through `0x5FFF` are registers: camera, clip (read only), draw color, fill pattern, the three palette remaps, and the page numbers. The bytes on both sides of that hole, 65280 of them, are the pixel window.

When the screen fits in those 65280 bytes, 64×64 and 128×128 do and 256×256 does not, the screen starts at address 0, row by row. What is left is the sprite sheet selected by `0x5F41` (0 is sheet 0). If that sheet is bigger than the space left, `0x5F42` picks the page and the map is not in the window. If the sheet fits, the tail is map cells from the map selected by `0x5F43`, one or two bytes each, and `0x5F44` pages the map when it does not fit in the tail.

When the screen does not fit, the whole window is one page of 65280 pixels. `poke(0x5F40, n)` turns the screen page. `poke(0x5F41, n)` with n other than 0 shows sheet n instead, and sheet 0 has no view in this mode. `poke(0x5F43, n)` with n other than 0 shows map n, and that wins over the sheet. Map 0 in this mode is only `mget` and `mset`.

A `peek` that does not land on a pixel or a register returns 0. A `poke` there does nothing. Sprite flags, SFX, and DATA tables are not in this window. Use `fget`, `sfxget`, and `dget`.

Practical use

Every function is listed at the bottom of this page, with practical uses of PIX37 functions. The links under each example open more about the functions.

Smooth movement

A 128×128 cart, seen from above. Sprite 1 is already drawn. Arrows accelerate in any direction, and friction slows you down. The sprite wraps: a half that leaves the bottom is drawn at the top, and the same for the other edges.

`_init _update _draw btn mid cls spr palt`

Checking if the next move is blocked

A 64×64 cart. The floor and the right-hand wall are already on the map, painted with sprite 3, and flag 0 is set on that sprite. Arrows move sprite 1. The word at the top says blocked when the next step would overlap a solid tile, and free when it would not.

`flr fget mget btn cls map spr print _update _draw`

A wipe with flip

Press A. The wipe is drawn by flip(), one black row at a time, instead of inside _draw(). _draw stays out of the way while that runs, so each row stays on screen.

`rectfill flip btnd cls print _update _draw`

A table of functions (Class)

Sprite 1 is already drawn. make_monster works like a class. It returns a table, and update and draw are methods that keep that monster's own x and y. The number at the top is how far it has walked.

`spr cls print flr _init _update _draw`

Four ways to read a button

A is button 4. Hold it and charge climbs. Release it and charge goes back to 0. Up and down repeat while held, so the menu row keeps moving. btnd is the first frame of a press, which this cart uses only as the start of that hold.

`btn btnd btnp btnu cls print _update _draw`

Shoot with the mouse

Sprite 1 is already drawn in the middle. A white dot is the cursor. Left click fires a pixel toward it. The wheel changes the speed, printed in the corner.

`scroll clickl mouse sqrt add all cls spr pset print _update _draw`

Type a name and confirm it

Press A to open the field. Up and down change a blinking letter. A writes it, X deletes, Y is space, and B confirms. The line then says your name.

```
btnd keyon keystr flr time cls print _update _draw
```

Camera on the player

A 128×128 cart. Sprite 1 starts in the middle of a floor of tiles. Arrows move. The camera keeps the sprite centered. x and y in the corner do not scroll, because camera() with no arguments puts the view back for the HUD.

```
btn screen camera cls map spr print flr _update _draw
```

A few lines of a visual novel

The lines are in DATA, in English and Polish. A continues, or picks stay when a choice is on screen. B picks leave on a choice, and otherwise swaps the language. X swaps the language while a choice is showing. The dialogue uses the 8px font so Polish letters draw.

```
dstr dget btnd cls print8 print _update _draw
```

Paint with peek and poke

A 64×64 cart, so address 0 is the top-left pixel and the next row starts at 64. The cart writes pixels, copies a row, clears a row, and prints the values it read back.

```
cls poke peek poke2 peek2 poke4 peek4 memcpy memset print _draw
```

A list you can add to

A adds a number, up to 8. B removes the last. X removes the first. The list is printed down the screen.

```
btnd count add deli cls ipairs print pack unpack next foreach select pcall assert del _update _draw
```

A name taken apart

The cart splits ada,12,blue and prints the name, its first letter, 12, and the word blue.

```
cls split sub chr ord tonum tostr print _draw
```

Play a note, then change it

Pattern 0 is one note long, and every replay is the same loudness. A plays it again when that note has finished. B raises the pitch and plays that note.

```
sfxset sfxhdr sfx btnd sfxstat sfxget cls print _init _update _draw
```

A song written into the cart

Two patterns loop on row 0, one on each of the first two channels. row stays 0. ch0 and ch1 are the note each channel is on. Editing the pattern, LEN, and LOOP in the cart is a way to see what those controls do in the music panel.

```
songget songset music musicstat cls print _init _draw
```

Three monsters in DATA

The DATA panel already has a name column and an hp column, with slime, bat, and eye. A hurts the slime. The numbers on the right are its hp.

```
drows dstr dget add btnd dset dcols cls print all _init _update _draw
```

Remember the score

A adds a point and saves it. Quit and open the cart again: the number is still there. B forgets it.

```
load btnd save delsave cls print _init _update _draw
```

A second chapter in another file

Two carts share the save slot quest. Part 1: A marks the game finished. Part 2, opened first, says to finish part 1. After that, part 2 opens, and B erases the save so part 1 can be played again.

```
load saveslot btnd save delsave cls print _init _update _draw
```

One achievement

Sprite 8 is already drawn. Press A ten times. The toast names the achievement, and sprite 8 stays because it is unlocked. B clears that achievement in the local file and resets the count.

```
achdef btnd achieved achieve achclear cls print spr _init _update _draw
```

A room you can walk in

A 256×256 cart. A white dot is the cursor. The menu is host, join, or the server list. Host opens a room named walk and starts the game. Join scrolls an address (digits, a dot, a colon) and B connects. The server list moves with up and down, and A joins the row. In the game, arrows walk sprite 1. One coin is shared: the host moves

it, and picking it up adds 1 to that player's score and spawns the next. Every sprite, including yours, shows that score beside it. The corner shows how many players are in, the LAN address, and the host's public address. B leaves: you and the others return to the menu, and the next host starts a new room at score 0.

`btn` `btnd` `mouse` `pset` `spr` `print` `flr` `rnd` `pairs` `nlisten` `nhost` `nannounce` `nlist` `njoin` `nleave` `nsend` `nkb` `nkbstr` `whoami` `nplayers` `nlan` `_mp` `_mp_join` `_mp_leave` `_mp_fail` `palt` `cls` `time` `_init` `_update` `_draw`



Dialogue that waits

A shows the next line. The function pauses at each yield until you press A again.

`yield` `cocreate` `btnd` `coresume` `cls` `print` `_init` `_update` `_draw`



Wait half a second

The word changes from ... to hi after about half a second. The screen keeps drawing while the coroutine is paused.

`yield` `cocreate` `costatus` `coresume` `cls` `print` `_init` `_update` `_draw`



Circles that arrive one by one

Five circles drip in from the top, two frames apart, and then fall.

`add` `yield` `cocreate` `coresume` `all` `cls` `circfill` `_init` `_update` `_draw`



Two frames, from the clock

Sprite 1 and sprite 2 are two frames of the same character. The picture swaps four times a second.

`cls` `flr` `time` `spr` `_draw`

Every function

Arguments in square brackets are optional. Indexes of tables and strings start at 1, except DATA rows and columns, which start at 0. Colors are palette indexes.

Index

`_draw`
`_init`
`_mp`
`_mp_fail`
`_mp_join`
`_mp_leave`
`_mp_update`
`_set_fps`
`_update`
`abs`
`achclear`
`achdef`
`achieve`
`achieved`
`add`

all
assert
atan2
band
bnot
bor
btn
btnd
btnp
btnu
bxor
camera
ceil
chr
circ
circfill
clickl
clickm
clickr
clip
cls
cocreate
color
coresume
cos
costatus
count
cursor
dcols
del
deli
delsave
dget
drows
dset
dstr
fget
fillp
flip
flr
foreach
fset
info
ipairs
keyoff
keyon
keystat
keystr
line
load
log
lshr
map
max
memcpy
memset
mget
mid
min
mouse
mset
music
musicstat
nannounce
ncall
next
nhost
njoin
nkb
nkbstr
nlan
nleave
nlist
nlisten
note
nplayers
nsend
nstun
ord
oval
ovalfill
pack
pairs
pal
palrgb
palt
pcall
peek
peek2
peek4
perf
pget
poke
poke2
poke4
print
print10
print8
pset
rect
rectfill
rnd
rotl
rotr
rrect
rrectfill
save
saveslot
screen
scroll
select
setmap
setpalette
setspr
settex
setturn
sfx
sfxget
sfxhdr
sfxset
sfxstat
sget
sgn

shl	spr	time
showstats	sqrt	tline
shr	srand	tonum
sin	sset	tostr
songget	sspr	unpack
songset	stat	whoami
split	sub	yield

Lifecycle

_init

`_init() ... end`

Runs once when the cartridge starts. Code that sits outside any hook is also folded into _init.

```
_init()
  x = 64
  y = 40
end
```

_update

`_update() ... end`

Game logic. Called at 60 Hz, even when _draw is capped or the host is slower. Read buttons here.

```
_update()
  if btn(0) then x -= 1 end
  if btn(1) then x += 1 end
end
```

_draw

```
_draw([fps]) ... end
```

Draws one frame. With no argument, or 0, it runs as often as the host presents. Pass n (1..999) to cap drawing at n frames per second. _update stays at 60 Hz.

```
_draw(30)
  cls(1)
  spr(1, x, y)
end
```

_set_fps

```
_set_fps(n)
```

Changes the _draw cap while the game is running. 0 means uncapped. n is clamped to 0..999.

```
if btn(4) then _set_fps(60) end
```

flip

```
flip()
```

Presents the screen and yields until the next frame. Use it only if you are drawing your own frame loop. Do not busy-wait after it.

```
cls(0)
print("tick", 4, 4, 7)
flip()
```

time

```
time()
```

Seconds since _init.

```
print(time(), 4, 4, 7)
```

Input

btn

```
btn([b], [p])
```

1 while button b is held, otherwise 0. p is the player, 0..7. With no arguments it returns a bit mask of player 0. Buttons: 0 left, 1 right, 2 up, 3 down, 4 A, 5 B, 6 X, 7 Y, 8 R, 9 L, 10 select, 11 start.

```
if btn(4) then shoot() end
```

btnd

```
btnd(b, [p])
```

True only on the _update frame when button b goes down.

```
if btnd(5) then menu_open = not menu_open end
```

btnp

```
btnp([b], [p])
```

True on press, then again while held: 30 ticks of delay, then every 8 ticks. With no button it returns a mask.

```
if btnp(2) then cursor -= 1 end
```

btnu

```
btnu(b, [p])
```

True only on the frame the button is released.

```
if btnu(4) then charge = 0 end
```

mouse

```
mouse()
```

Returns mouse x, y in game pixels.

```
mx, my = mouse()
pset(mx, my, 7)
```

clickl

```
clickl([p])
```

Left mouse button. clickl() is held. clickl(1) is the press edge.

```
if clickl(1) then place(mouse()) end
```

clickr

```
clickr([p])
```

Right mouse button. Same 1-argument edge as clickl.

```
if clickr() then cancel() end
```

clickm

```
clickm([p])
```

Middle button, or a click of the scroll wheel.

```
if clickm(1) then drag = true end
```

scroll

`scroll()`

Mouse-wheel movement this frame. Positive and negative depend on the wheel direction.

```
cam_y -= scroll()
```

keyon

`keyon()`

Opens a text field driven by buttons. Up and down change a letter A–Z. A writes that letter. X deletes. Y is space. B confirms and closes the field. The keyboard does not type into it.

```
if btnd(4) then keyon() end
```

keyoff

`keyoff()`

Hides the text field.

```
keyoff()
```

keystr

`keystr([s])`

Text in the field, then 1 on the frame B confirms, then the letter up and down are changing. That letter is empty once the field closes. keystr("") clears the text.

```
name, done, letter = keystr()
if done == 1 then start(name) end
```

keystat

```
keystat()
```

1 while the text field is open, otherwise 0.

```
if keystat() == 1 then return end
```

Drawing

screen

```
screen()
```

Current cartridge screen size in pixels. Returns width, then height. This is the game resolution, not the editor window.

```
w, h = screen()
```

cls

```
cls([col])
```

Clears the screen. Default color is 0.

```
cls(1)
```

color

```
color(col)
```

Sets the draw color used when a later call omits col. With no argument it returns the current color.

```
color(7)
print("hi")
```

camera

`camera([x, y])`

Offsets every draw call. camera() resets to 0, 0.

```
camera(cam_x, cam_y)
```

clip

`clip([x, y, w, h, clip])`

Limits drawing to a rectangle. A nonzero fifth argument intersects with the current clip. clip() restores the full screen. Peek of the clip registers is read-only.

```
clip(8, 8, 48, 48)
circfill(32, 32, 20, 8)
clip()
```

cursor

`cursor([x, y, col])`

Sets where the next print without coordinates goes, and optionally the color. cursor() returns x, y.

```
cursor(4, 4, 7)
print("line 1")
print("line 2")
```

pset

`pset(x, y, [col])`

Sets one screen pixel. Counts toward the px stat.

```
pset(x, y, 10)
```

pget

```
pget(x, y)
```

Color of the screen pixel at x, y.

```
if pget(x, y+8) == 3 then grounded = true end
```

print

```
print(v, [x, y, col])
```

Draws 3×5 text (UTF-8, cropped from the Fusion font). Without x, y it prints at the cursor and moves to the next line.

```
print("score "..score, 2, 2, 7)
```

log

```
log(...)
```

Appends the arguments, separated by spaces, to a text buffer. pix37_run prints that buffer on stdout. It does not draw.

```
log("hp", hp)
```

print8

```
print8(v, [x, y, col])
```

Same as print, with the 8 px Fusion font.

```
print8("pause", 16, 28, 7)
```

print10

```
print10(v, [x, y, col])
```

Same as print, with the 10 px Fusion font.

```
print10("pix37", 8, 8, 10)
```

line

```
line([x0, y0, x1, y1, col])
```

Line from x0,y0 to x1,y1. Two arguments continue from the previous end. line() resets that chain.

```
line(0, 32, 63, 32, 6)
```

circ

```
circ(x, y, r, [col])
```

Circle outline. Radius is in pixels.

```
circ(32, 32, 12, 12)
```

circfill

```
circfill(x, y, r, [col])
```

Filled circle.

```
circfill(x, y, 3, 8)
```

oval

```
oval(x0, y0, x1, y1, [col])
```

Ellipse outline inside the bounding box. Corners are inclusive.

```
oval(8, 20, 56, 44, 7)
```

ovalfill

`ovalfill(x0, y0, x1, y1, [col])`

Filled ellipse in a bounding box.

```
ovalfill(10, 40, 54, 60, 3)
```

rect

`rect(x0, y0, x1, y1, [col])`

Rectangle outline. Corners are inclusive.

```
rect(0, 0, 63, 63, 7)
```

rectfill

`rectfill(x0, y0, x1, y1, [col])`

Filled rectangle.

```
rectfill(0, 56, 63, 63, 3)
```

rrect

`rrect(x, y, w, h, [r, col])`

Rounded rectangle outline. x, y is the top left. w and h are the size in pixels; both must be greater than 0. r is the corner radius and defaults to 0. It is clamped to half of the shorter side.

```
rrect(8, 8, 48, 16, 3, 7)
```

rrectfill

```
rrectfill(x, y, w, h, [r, col])
```

Filled rounded rectangle. Same arguments as rrect.

```
rrectfill(8, 40, 48, 16, 2, 1)
```

fillp

```
fillp([p])
```

16-bit fill pattern for later filled shapes. 0 is solid. fillp() returns the current pattern. Bits walk the 4×4 pattern.

```
fillp(0xA5A5)
rectfill(8, 8, 40, 40, 10)
fillp(0)
```

pal

```
pal([c0, c1, [mode]])
```

Remaps color c0 to c1. mode 0 is the draw palette, 1 the display palette, 2 the fill-pattern palette. pal() resets.

```
pal(8, 10)
circfill(32, 32, 8)
```

palrgb

```
palrgb(col, r, g, b)
```

Sets one palette entry to RGB, each channel 0..255.

```
palrgb(8, 255, 80, 40)
```

palt

`palt([col, t])`

Transparency. palt(col) or palt(col, true) makes col transparent. palt(col, false) makes it solid. palt() makes every color opaque.

```
palt(0)
spr(1, x, y)
```

setpalette

`setpalette(n)`

Selects palette bank n. setpalette() returns the current bank.

```
setpalette(1)
```

tline

`tline(x0, y0, x1, y1, mx, my, [mdx, mdy])`

Draws a line on the screen while sampling map pixels from map position mx, my. mdx, mdy step the map sample along the line.

```
tline(0, 40, 63, 20, 0, 0, 0.5, 0)
```

Sprites and maps

spr

`spr(n, x, y, [w, h, fx, fy, tex, rot])`

Draws sprite n at x, y. w and h are the size in pixels, default 8. fx and fy flip horizontally and vertically when nonzero. tex is the sprite-sheet tab. rot is the angle in turns (0..1), same direction as sin and cos. 0 leaves the sprite unrotated.

```
spr(1, x, y, 8, 8, 0, 0, 0, aim)
```

sspr

```
sspr(sx, sy, sw, sh, dx, dy, [dw, dh, fx, fy, tex])
```

Copies a rectangle of the sprite sheet to the screen, scaled to dw by dh. Defaults keep the source size.

```
sspr(0, 0, 8, 8, 16, 16, 32, 32)
```

sget

```
sget(x, y, [tex])
```

Color of one texel on the sprite sheet.

```
c = sget(0, 0)
```

sset

```
sset(x, y, [col, tex])
```

Writes one texel on the sprite sheet.

```
sset(3, 3, 8)
```

fget

```
fget(n, [f, tex])
```

Sprite flags. fget(n) returns the flag byte. fget(n, f) is true when flag bit f (0..7) is set.

```
if fget(tile, 0) then solid = true end
```

fset

```
fset(n, [f], v, [tex])
```

Sets sprite flags. fset(n, v) writes the whole byte. fset(n, f, v) sets one bit.

```
fset(2, 0, true)
```

setspr

```
setspr(n)
```

Selects the active sprite sheet. setspr() returns the current sheet. Same bank as settex.

```
setspr(1)
```

settex

```
settex(n)
```

Alias of setspr.

```
settex(0)
```

map

```
map(cx, cy, sx, sy, [cw, ch, layer, tex])
```

Draws the map. Cell cx, cy of the map is drawn at screen sx, sy. cw, ch of 0 means the whole map. layer is accepted and ignored. tex is the tileset sheet.

```
map(0, 0, 0, 0, 16, 16)
```

mget

```
mget(cx, cy)
```

Sprite id in map cell cx, cy.

```
if fget(mget(tx, ty), 0) then bounce() end
```

mset

```
mset(cx, cy, n)
```

Writes sprite n into map cell cx, cy.

```
mset(3, 4, 0)
```

setmap

```
setmap(n)
```

Selects map n. setmap() returns the current map. Map 0 is a real map.

```
setmap(1)
```

Memory

peek

```
peek(addr)
```

Reads one byte of the 64 KB window. The window is the live screen, then sprite pixels, then map cells when the screen fits in 65280 bytes. Larger screens are paged with poke(0x5F40, page). Addresses that are not mapped return 0. Also written @addr.

```
px = peek(0)
```

poke

```
poke(addr, ...)
```

Writes bytes at addr. Extra arguments write the following bytes. Unmapped addresses are ignored. Also written $addr, val. Poking the screen counts as a pixel write.

```
poke(0x5F00, 4) -- camera x
```

peek2

`peek2(addr)`

Unsigned 16-bit value, little-endian, at addr.

```
w = peek2(0x6000)
```

poke2

`poke2(addr, val)`

Writes an unsigned 16-bit value, little-endian.

```
poke2(0x6000, 1000)
```

peek4

`peek4(addr)`

Unsigned 32-bit value, little-endian.

```
n = peek4(0x6000)
```

poke4

`poke4(addr, val)`

Writes an unsigned 32-bit value, little-endian.

```
poke4(0x6000, 1)
```

memcpy

`memcpy(dst, src, len)`

Copies len bytes inside the same 64 KB window.

```
memcpy(0x6000, 0x6100, 64)
```

memset

`memset(dest, val, len)`

Fills len bytes starting at dest with val.

```
memset(0, 0, 64)
```

Math

abs

`abs(x)`

Absolute value.

```
d = abs(tx - x)
```

flr

`flr(x)`

Rounds toward minus infinity.

```
tx = flr(x / 8)
```

ceil

`ceil(x)`

Rounds toward plus infinity.

```
n = ceil(hp / 8)
```

max

`max(a, b)`

Larger of the two values.

```
x = max(0, x)
```

min

`min(a, b)`

Smaller of the two values.

```
x = min(63, x)
```

mid

`mid(a, b, c)`

Middle of the three values. Useful as a clamp.

```
x = mid(0, x, 63)
```

sgn

`sgn(x)`

Sign of x: -1, 0, or 1.

```
x += sgn(dir)
```

sqrt

`sqrt(x)`

Square root. A negative argument is not a number.

```
dist = sqrt(dx*dx + dy*dy)
```

sin

`sin(x)`

Sine. x is in turns: 0, 0.25, 0.5, 0.75 go around the circle once. Not radians.

```
y = 32 + sin(time()) * 8
```

cos

`cos(x)`

Cosine, in turns, same convention as sin.

```
x = 32 + cos(time()) * 8
```

atan2

`atan2(dx, dy)`

Angle of the vector dx, dy, in turns from 0 to 1.

```
aim = atan2(mx - x, my - y)
```

rnd

`rnd([x])`

Random number in [0, x). Default x is 1, so rnd() is 0..1. For a die, flr(rnd(6))+1.

```
c = flr(rnd(16))
```

srand

`srand(seed)`

Seeds the random generator so a later rnd sequence can be repeated.

```
srand(1)
```

band

```
band(a, b)
```

Bitwise AND. Also written a & b.

```
if band(flags, 1) ~= 0 then hit() end
```

bor

```
bor(a, b)
```

Bitwise OR. Also written a | b.

```
flags = bor(flags, 4)
```

bxor

```
bxor(a, b)
```

Bitwise XOR. Also written a ^^ b.

```
n = bxor(n, 255)
```

bnot

```
bnot(a)
```

Bitwise NOT. Also written ~a.

```
mask = bnot(1)
```

shl

```
shl(a, n)
```

Shift left. Also written a << n.

```
n = shl(1, bit)
```

shr

`shr(a, n)`

Arithmetic shift right. Also written a >> n.

```
n = shr(n, 1)
```

lshr

`lshr(a, n)`

Logical shift right. Also written a >>> n.

```
n = lshr(n, 4)
```

rotl

`rotl(a, n)`

Rotate bits left. Also written a <<> n.

```
n = rotl(n, 1)
```

rotr

`rotr(a, n)`

Rotate bits right. Also written a >>< n.

```
n = rotr(n, 1)
```

Tables and values

add

`add(t, v, [i])`

Appends v to table t. If i is given, inserts at that 1-based index.

```
add(bullets, {x=x, y=y})
```

del

`del(t, v)`

Removes the first entry equal to v.

```
del(items, gem)
```

deli

`deli(t, [i])`

Removes index i and returns that value. With no i, removes the last entry.

```
local b = deli(bullets, i)
```

all

`all(t)`

Iterator over the array part, in order. Holes stop the walk the way a 1-based array ends.

```
for b in all(bullets) do
  b.x += 2
end
```

foreach

```
foreach(t, fn)
```

Calls fn(v) for each array element.

```
foreach(enemies, draw_enemy)
```

pairs

```
pairs(t)
```

Iterator over every key and value, including named fields.

```
for k, v in pairs(player) do
  print(k)
end
```

ipairs

```
ipairs(t)
```

Iterator over the array part with the 1-based index.

```
for i, v in ipairs(deck) do
  spr(v, i*8, 8)
end
```

next

```
next(t, [k])
```

Returns the next key and value after k. next(t) starts at the beginning. Used by pairs.

```
k, v = next(t)
```

count

```
count(t, [v])
```

Length of the array part. If v is given, counts how many entries equal v.

```
if count(gems) == 0 then win() end
```

pack

```
pack(...)
```

Packs the arguments into a new table.

```
t = pack(1, 2, 3)
```

unpack

```
unpack(t, [i, j])
```

Returns t[i] through t[j] as separate values. Defaults cover the whole array part.

```
a, b = unpack(t)
```

select

```
select(i, ...)
```

select("#", ...) returns how many arguments follow. select(i, ...) returns the arguments from i onward.

```
n = select("#", 1, 2, 3)
```

assert

```
assert(c, [msg])
```

Stops with an error when c is false or nil. msg replaces the default text.

```
assert(hp > 0, "dead")
```

pcall

```
pcall(f, ...)
```

Calls f. On success returns true and the results. On error returns false and the error text.

```
ok, err = pcall(load_level, 3)
```

Strings

chr

```
chr(n)
```

One character from a byte code 0..255.

```
s = chr(65) -- A
```

ord

```
ord(s, [i, j])
```

Byte codes of s from index i to j. Indexes are 1-based. Defaults to the first character.

```
code = ord("a")
```

sub

```
sub(s, i, [j])
```

Substring from i to j, 1-based. Omit j to take the tail.

```
name = sub(keystr(), 1, 8)
```

split

```
split(s, [sep, conv])
```

Splits s into a table. sep defaults to comma. A number sep is a chunk width. conv converts pieces with tonum when it is true.

```
words = split("red,blue", ",")
```

tonum

```
tonum(s, [base])
```

Parses a number. base may be 2..36. Returns nil when the text is not a number.

```
n = tonum("255")
```

tostr

```
tostr(v, [digits])
```

Turns a value into text. digits sets decimal places for numbers.

```
print(tostr(hp, 0), 4, 4, 7)
```

Audio

sfx

```
sfx(n, [ch, offset, len])
```

Plays pattern n and returns the channel. Omit ch, or pass -1, and the first free channel is chosen. ch -2 stops pattern n on every channel. n -1 stops a channel, or every channel when ch is omitted. n -2 lets a looping pattern finish instead of looping. sfx() with no arguments returns -1.

```
ch = sfx(0)
```

sfxget

```
sfxget(n, step)
```

Returns a table of one pattern step: [1] pitch, [2] wave, [3] volume, [4] effect. Pitch is 0..63 (C-0 through D#5). Wave is 0..15. Volume is 0..7, and 0 is silent. Effect is 0..7.

```
step = sfxget(0, 0)
pitch = step[1]
```

sfxset

```
sfxset(n, step, pitch, wave, vol, fx)
```

Writes one pattern step. All six arguments are required.

```
sfxset(0, 0, 24, 3, 5, 0)
```

sfxhdr

```
sfxhdr(n, [key, val])
```

Pattern header. sfxhdr(n) returns the header table. sfxhdr(n, key, val) sets one field: speed, loop_in, loop_out, level, noise, crush, dtune, rverb, dampn, swing, steps. Numbers 0..10 select the same fields in that order. speed is TIC, ticks per step. Higher is slower. 15 is 120 beats per minute. 16-bit accepts 1..255. Narrower synths snap to their TIC list.

```
sfxhdr(0, "speed", 15)
```

sfxstat

```
sfxstat([ch])
```

Pattern id playing on channel ch, or -1. Default channel is 0.

```
if sfxstat(0) < 0 then sfx(1) end
```

note

```
note(pitch, [wave, vol, fx, ch, ticks])
```

Plays one note without a pattern. pitch 0..63, wave 0..15, volume 0..7, effect 0..7. ticks is how long it holds, in the same 120 Hz ticks as TIC. Omit ticks and it lasts 15, one default step. The effect runs across that time, then the channel goes quiet.

```
note(36, 3, 5, 5, 0)
```

music

```
music([n, fade, mask])
```

Starts song n from the top. music(-1) stops. Calling music with the id that is already playing leaves it running. music() returns the playing song id, or -1. fade is milliseconds: the song fades in, or fades out when n is -1. mask bits that are set are the channels used for the song. 0 means all eight. The other channels stay free for sfx.

```
music(0, 1000, 7)
```

musicstat

```
musicstat()
```

Table of the playing song. song and row are the song id and the row, or -1 when nothing is playing. step and pos have eight numbers. step[1] is channel 0, step[8] is channel 7. step is the note index, or -1 when that channel is silent. pos is how far that note has gone, from 0 to 1. tick is how many ticks this row has already played. played is how many rows have started since music(), including the one that is playing.

```
st = musicstat()
row = st.row
note = st.step[1]
```

songget

songget(song, [row, ch])

Pattern id stored in a song cell. ch is 0..7. 255 means the cell is empty. This is the live call; the code hint text still shows an older argument order.

```
id = songget(0, 0, 0)
```

songset

songset(song, row, ch, pattern)

Writes a pattern id into a song cell. Use 255 to clear the cell.

```
songset(0, 0, 0, 1)
```

stat

stat(n)

Audio only. 16..23 is the sfx id on channel n-16. 24..31 is the music pattern on channel n-24. Any other n returns 0. Frame timing and RAM are perf and info, not stat.

```
id = stat(16)
```

Data tables

dget

dget(t, row, col)

Reads a cell of data table t from the DATA panel. row and col start at 0.

```
hp = dget(0, 0, 1)
```

dset

```
dset(t, row, col, v)
```

Writes a cell. Use a string for a text column.

```
dset(0, 0, 0, "slime")
```

dstr

```
dstr(t, row, col)
```

The cell as text. Numbers are converted.

```
name = dstr(0, row, 0)
```

drows

```
drows(t)
```

How many rows table t has.

```
for r = 0, drows(0) - 1 do
end
```

dcols

```
dcols(t)
```

How many columns table t has.

```
cols = dcols(0)
```

Save and achievements

save

`save(k, v)`

Stores v under key k in the save file. v may be a number, a string, or a table of those. The file is compressed.

```
save("hp", hp)
```

load

`load(k)`

Reads key k. Returns nil when the key is missing.

```
hp = load("hp") or 3
```

delsave

`delsave(k)`

Deletes key k from the save.

```
delsave("hp")
```

saveslot

`saveslot([name])`

Chooses the save file inside the author's folder. With no name, the slot is the game title. Returns 1 when that file already exists.

```
if saveslot("slot1") == 1 then
  hp = load("hp") or 3
end
```

achieve

```
achieve(i)
```

Unlocks achievement i, from 0 to 15. Stored locally. A Steam build also sets ACH_i. A catalog session reports it to the player's account.

```
if coins >= 100 then achieve(0) end
```

achieved

```
achieved(i)
```

1 when achievement i is already unlocked.

```
if achieved(0) == 1 then spr(8, 4, 4) end
```

achdef

```
achdef(i, text, [spr, tex])
```

Toast text for achievement i, shown for 15 seconds from the bottom of the screen. spr is an optional 8×8 tile, tex its sheet.

```
achdef(0, "rich", 8)
```

achclear

```
achclear(i)
```

Clears achievement i in the local file. A Steam ACH_i that was already set stays set.

```
if btnd(5) then achclear(0) end
```

Network

whoami

```
whoami()
```

Your player id, 0..15, while a session is up. Offline it is -1.

```
id = whoami()
```

nhost

```
nhost()
```

1 if this machine is the host, otherwise 0.

```
if nhost() == 1 then spawn() end
```

nplayers

```
nplayers()
```

How many live ids exist in 0..15.

```
print(nplayers(), 4, 4, 7)
```

nlisten

```
nlisten()
```

Starts hosting UDP and returns the LAN address as ip:port.

```
addr = nlisten()
```

njoin

```
njoin(addr)
```

Joins ip or ip:port.

```
njoin("10.0.0.8:7777")
```

nleave

```
nleave()
```

Leaves the session. Peers see _mp_leave with why 0.

```
if btnd(11) then nleave() end
```

nsend

```
nsend([p,] name, ...)
```

Sends a lossy message. Name is the event. Up to four extra values follow. p from 0 to 15 whispers to that peer; omit p to send to everyone.

```
nsend("pos", x, y)
```

ncall

```
ncall([p,] name, ...)
```

Same shape as nsend, but the message is retried until acknowledged. Three failures in a row kill the session.

```
ncall("hit", target, dmg)
```

nlan

```
nlan()
```

Current LAN ip:port after nlisten.

```
print(nlan(), 2, 2, 7)
```

nstun

```
nstun()
```

Your public ip:port from STUN. A guest on another network can try to join it.

```
pub = nstun()
```

setturn

```
setturn(addr, [user, pass])
```

TURN fallback when a direct path fails. setturn() with empty arguments turns it off. The address is host:port.

```
setturn("turn.example:3478", user, pass)
```

nlist

```
nlist(game)
```

Table of open sessions for that game name. Each entry has addr, players, and name.

```
rooms = nlist("duel")
```

nannounce

```
nannounce(game)
```

Heartbeat so this host shows up in nlist for game. Call it on a timer while hosting.

```
nannounce("duel")
```

nkb

`nkb([on])`

Address field. nkb(1) opens it, nkb(0) hides it, nkb() tells you if it is open. Up and down change a character, 0–9 or a dot or a colon. A writes it. X deletes. The keyboard does not type into it.

```
nkb(1)
```

nkbstr

`nkbstr([s])`

Text in the address field, then the character up and down are changing. nkbstr("") clears the text.

```
addr, digit = nkbstr()
```

_mp

`_mp(from, name, ...)`

You write this function. It runs when a message arrives. from is the sender id. name is the event. Up to four values follow.

```
function _mp(from, name, a, b)
  if name == "pos" then
    px[from] = a
    py[from] = b
  end
end
```

_mp_join

`_mp_join(p)`

You write this function. A peer received id p. It also fires for you; whoami() is that id.

```
function _mp_join(p)
  add(log, p)
end
```

_mp_leave

```
_mp_leave(p, why)
```

You write this function. Peer p left. why 0 is a normal leave. why 1 is a lost connection.

```
function _mp_leave(p, why)
  px[p] = nil
end
```

_mp_update

```
_mp_update()
```

You write this function. It runs every 15 _update frames, about 4 Hz. A good place to nsend positions.

```
function _mp_update()
  nsend("pos", x, y)
end
```

_mp_fail

```
_mp_fail()
```

You write this function. The session is dead: the host disappeared, or ncall failed three times.

```
function _mp_fail()
  online = false
end
```

Coroutines

cocreate

```
cocreate(f)
```

Makes a coroutine from function f. It does not run until coresume.

```
co = cocreate(cutscene)
```

coresume

```
coresume(c, ...)
```

Runs coroutine c until it yields or returns. On success: true and the yielded values. On error: false and the error.

```
ok, a = coresume(co)
```

costatus

```
costatus(c)
```

"suspended", "running", or "dead".

```
if costatus(co) == "dead" then co = nil end
```

yield

```
yield(...)
```

Gives control back to whoever called coresume. The values become the results of that coresume.

```
yield(1)
```

Measure

info

```
info([x, y, col])
```

Draws the live HUD in _draw: fps (1 second average), upd, drw, px, max, ram, peak, ramc, and net when a session is up. info() uses the print cursor. info(col) sets only the color. info(x, y) or info(x, y, col) places it. Numbers use B, k, or m so they fit a small screen. The call itself is not counted in px.

```
info(1, 1, 7)
```

showstats

```
showstats([on])
```

Same HUD as info, drawn automatically at 0, 0 after _draw, ignoring camera and clip. showstats(0) hides it. showstats(1) or showstats() shows it.

```
showstats(1)
```

perf

```
perf(n)
```

One HUD number, so you can draw it yourself. 0 fps, 1 upd, 2 drw, 3 px, 4 max (heaviest frame this second), 5 ram, 6 ramc (ram plus the packed cartridge), 7 peak ram since the cart started.

```
print(perf(0), 2, 2, 7)
```