PICO-8 User Manual

PICO-8 v0.2.6
https://www.pico-8.com

Author: Joseph White // hey@lexaloffle.com

PICO-8 is built with:

- SDL2 http://www.libsdl.org
- Lua 5.2 http://www.lua.org // see license.txt
- ws281x by jgarff // see license.txt
- GIFLIB http://giflib.sourceforge.net/
- WiringPi http://wiringpi.com/
- libb64 by Chris Venter
- miniz by Rich Geldreich
- z8lua by Sam Hocevar https://github.com/samhocevar/z8lua

Latest version of this manual (as html, txt) and other resources:

https://www.lexaloffle.com/pico-8.php?page=resources

Welcome to PICO-8!

PICO-8 is a fantasy console for making, sharing and playing tiny games and other computer programs. When you turn it on, the machine greets you with a shell for typing in Lua programs and provides simple built-in tools for creating sprites, maps and sound.

The harsh limitations of PICO-8 are carefully chosen to be fun to work with, encourage small but expressive designs and hopefully to give PICO-8 cartridges their own particular look and feel.

Specifications

```
Display: 128x128, fixed 16 colour palette
Input:  6-button controllers
Carts:  32k data encoded as png files
Sound:  4 channel, 64 definable chip blerps
Code:  P8 Lua (max 8192 tokens of code)
CPU:   4M vm insts/sec
Sprites: Single bank of 128 8x8 sprites (+128 shared)
Map:   128 x 32 Tilemap (+ 128 x 32 shared)
```

❶ Getting Started

1.1 Keys

```
ALT+ENTER: Toggle Fullscreen
ALT+F4:    Fast Quit (Windows)
CTRL-Q:    Fast Quit (Mac, Linux)
CTRL-R:    Reload / Run / Restart cartridge
CTRL-S:    Quick-Save working cartridge
CTRL-M:    Mute / Unmute Sound
ENTER / P: Pause Menu (while running cart)
```

```
Player 1 default keys: Cursors + ZX / NM / CV
Player 2 default keys: SDFE + tab,Q / shift A
```

To change the default keys use the KEYCONFIG utility from inside PICO-8:

```
> KEYCONFIG
```

1.2 Hello World

After PICO-8 boots, try typing some of these commands followed by enter:

```
> PRINT("HELLO WORLD")
> RECTFILL(80,80,120,100,12)
> CIRCFILL(70,90,20,14)
> FOR I=1,4 DO PRINT(I) END
```

(Note: PICO-8 only displays upper-case characters -- just type normally without capslock!)

You can build up an interactive program by using commands like this in the code editing mode along with two special callback functions **_UPDATE** and **_DRAW**. For example, the following program allows you to move a circle around with the cursor keys. Press Esc to switch to the code editor and type or copy & paste the following code:

```
X = 64 Y = 64
FUNCTION _UPDATE()
 IF (BTN(0)) THEN X=X-1 END
 IF (BTN(1)) THEN X=X+1 END
 IF (BTN(2)) THEN Y=Y-1 END
 IF (BTN(3)) THEN Y=Y+1 END
END

FUNCTION _DRAW()
 CLS(5)
 CIRCFILL(X,Y,7,14)
END
```

Now press Esc to return to the console and type RUN (or press CTRL-R) to see it in action. Please refer to the demo cartridges for more complex programs (type INSTALL_DEMOS).

If you want to store your program for later, use the SAVE command:

```
> SAVE PINKCIRC
```

And to load it again:

```
> LOAD PINKCIRC
```

1.3 Example Cartridges

These cartridges are included with PICO-8 and can be installed by typing:

```
> INSTALL_DEMOS
> CD DEMOS
> LS

HELLO □ □ □Greetings from PICO-8
API □ □ □ □Demonstrates most PICO-8 functions
JELPI □ □ □Platform game demo w/ 2p support
CAST □ □ □ 2.5D Raycaster demo
DRIPPY □ □ Draw a drippy squiggle
WANDER □ □ Simple walking simulator
COLLIDE □ □Example wall and actor collisions
```

To run a cartridge, open PICO-8 and type:

```
> LOAD JELPI
> RUN
```

Press escape to stop the program, and once more to enter editing mode.

A small collection of BBS carts can also be installed to /GAMES with:

```
> INSTALL_GAMES
```

1.4 File System

These commands can be used to manage files and directories (folders):

```
LS □ □ □ □ □list the current directory
CD BLAH □ □ change directory
CD .. □ □ □ go up a directory
CD / □ □ □ □change back to top directory (on PICO-8's virtual drive)
MKDIR BLAH □make a directory
FOLDER □ □ □open the current directory in the host operating system's file browser
```

LOAD BLAH load a cart from the current directory SAVE BLAH save a cart to the current directory

If you want to move files around, duplicate them or delete them, use the FOLDER command and do it in the host operating system.

The default location for PICO-8's drive is:

Windows: C:/Users/Yourname/AppData/Roaming/pico-8/carts
OSX: /Users/Yourname/Library/Application Support/pico-8/carts
Linux: ~/.lexaloffle/pico-8/carts

You can change this and other settings in pico-8/config.txt

☆ Tip: The drive directory can be mapped to a cloud drive (provided by Dropbox, Google Drive or similar) in order to create a single disk shared between PICO-8 machines spread across different host machines.

1.5 Loading and Saving

When using LOAD and SAVE, the .P8 extension can be omitted and is added automatically.

```
> SAVE FOO
SAVED FOO.P8
```

Cartridge files can also be dragged and dropped into PICO-8's window to load them.

Using .p8.png filename extension will write the cartridge in a special image format that looks like a cartridge. Using .p8.rom" writes in a raw 32k binary format.

Use a filename of "@clip" to load or save to the clipboard.

Use a filename of "@url" to save to clipboard as a pico-8-edu.com url if it can be encoded in 2040 characters (code and gfx only).

Once a cartridge has been loaded or saved, it can also be quick-saved with CTRL-S

Saving .p8.png carts with a text label and preview image

To generate a label image saved with the cart, run the program first and press CTRL-7 to grab whatever is on the screen. The first two lines of the program starting with '--' are also drawn to the cart's label.

```
-- OCEAN DIVER LEGENDS
-- BY LOOPY
```

Code size restrictions for .p8.png / .p8.rom formats

When saving in .png or .rom format, the compressed size of the code must be less than 15360 bytes so that the total data is <= 32k.

To find out the current size of your code, use the INFO command. The compressed size limit is not enforced for saving in .p8 format.

1.6 Using an External Text Editor

It is possible to edit .p8 files directly with a separate text editor. Using CTRL-R to run a cartridge will automatically re-load the file if:

1. There are no unsaved changes in the PICO-8 editors, AND
2. The file differs in content from the last loaded version.

If there are changes to both the cart on disk and in the editor, a notification is displayed:

```
DIDN'T RELOAD; UNSAVED CHANGES
```

Alternatively, .lua text files can be modified in a separate editor and then included into the cartridge's code each time it is run using **#INCLUDE** (in the desired code location):

```
#INCLUDE YOURFILE.LUA
```

1.7 Backups

When quitting without saving changes, or overwriting an existing file, a backup of the cartridge is saved to {appdata}/pico-8/backup. An extra copy of the current cartridge can also be saved to the same folder by typing BACKUP.

To open the backups folder in the host operating system's file browser, use:

```
> FOLDER BACKUPS
```

It is then possible to drag and drop files into the PICO-8 window to load them.

From 0.2.4c, periodic backups are also saved every 20 minutes when not idle in the editor, which means the backups folder will grow by about 1MB every 5 hours. This can be disabled or adjusted in config.txt

1.8 Configuration

PICO-8 reads configuration settings from config.txt at the start of each session, and saves it on exit (so you should edit config.txt when PICO-8 is not running).

The location of the config.txt file depends on the host operating system:

Windows: C:/Users/Yourname/AppData/Roaming/pico-8/config.txt
OSX: /Users/Yourname/Library/Application Support/pico-8/config.txt
Linux: ~/.lexaloffle/pico-8/config.txt

Use the -home switch (below) to use a different path to store config.txt and other data.

Some settings can be changed while running PICO-8 by typing CONFIG SETTING VALUE. (type CONFIG by itself for a list)

Commandline parameters

```
// note: these override settings found in config.txt

pico8 [switches] [filename.p8]
```

```
-width n □ □ □ □ □ □ □ □set the window width
-height n □ □ □ □ □ □ □ set the window height
-windowed n □ □ □ □ □ □ set windowed mode off (0) or on (1)
-volume n □ □ □ □ □ □ □ set audio volume 0..256
-joystick n □ □ □ □ □ □ joystick controls starts at player n (0..7)
-pixel_perfect n □ □ □ □1 for unfiltered screen stretching at integer scales (on by
default)
-preblit_scale n □ □ □ □scale the display by n before blitting to screen (useful with -
pixel_perfect 0)
-draw_rect x,y,w,h □ □ □absolute window coordinates and size to draw pico-8's screen
-run filename □ □ □ □ □ load and run a cartridge
-x filename □ □ □ □ □ □ execute a PICO-8 cart headless and then quit (experimental!)
-export param_str □ □ □ run EXPORT command in headless mode and exit (see notes under
export)
-p param_str □ □ □ □ □ □pass a parameter string to the specified cartridge
-splore □ □ □ □ □ □ □ □ boot in splore mode
-home path □ □ □ □ □ □ □set the path to store config.txt and other user data files
-root_path path □ □ □ □ set the path to store cartridge files
-desktop path □ □ □ □ □ set a location for screenshots and gifs to be saved
-screenshot_scale n □ □ scale of screenshots. □default: 3 (368x368 pixels)
-gif_scale n □ □ □ □ □ □scale of gif captures. default: 2 (256x256 pixels)
-gif_len n □ □ □ □ □ □ □set the maximum gif length in seconds (1..120)
-gui_theme n □ □ □ □ □ □use 1 for a higher contrast editor colour scheme
-timeout n □ □ □ □ □ □ □how many seconds to wait before downloads timeout (default: 30)
-software_blit n □ □ □ □use software blitting mode off (0) or on (1)
-foreground_sleep_ms n □how many milliseconds to sleep between frames.
-background_sleep_ms n □how many milliseconds to sleep between frames when running in
background
-accept_future n □ □ □ □1 to allow loading cartridges made with future versions of
PICO-8
-global_api n □ □ □ □ □ 1 to leave api functions in global scope (useful for debugging)
```

□ Controller Setup

PICO-8 uses the SDL2 controller configuration scheme. It will detect common controllers on startup and also looks for custom mappings in sdl_controllers.txt in the same directory as config.txt. sdl_controllers.txt has one mapping per line.

To generate a custom mapping string for your controller, use either the controllermap program that comes with SDL2, or try http://www.generalarcade.com/gamepadtool/

To find out the id of your controller as it is detected by SDL2, search for "joysticks" or "Mapping" in log.txt after running PICO-8. This id may vary under different operating systems. See: https://www.lexaloffle.com/bbs/?tid=32130

To set up which keyboard keys trigger joystick buttons presses, use KEYCONFIG.

1.9 Screenshots and GIFS

While a cartridge is running use:

```
CTRL-6 Save a screenshot to desktop
CTRL-7 Capture cartridge label image
CTRL-8 Start recording a video
CTRL-9 Save GIF video to desktop (8 seconds by default)
```

You can save a video at any time (it is always recording); CTRL-8 simply resets the video starting point. To record more than 8 seconds, use the CONFIG command (maximum: 120)

```
CONFIG GIF_LEN 60
```

If you would like the recording to reset every time (to create a non-overlapping sequence), use:

```
CONFIG GIF_RESET_MODE 1
```

The gif format can not match 30fps exactly, so PICO-8 instead uses the closest match: 33.3fps.

If you have trouble saving to the desktop, try configuring an alternative desktop path in config.txt

1.10 Sharing Cartridges

There are three ways to share carts made in PICO-8:

1. Share the .p8 or .p8.png file directly with other PICO-8 users

Type FOLDER to open the current folder in your host operating system.

2. Post the cart on the Lexaloffe BBS to get a web-playable version

http://www.lexaloffle.com/pico-8.php?page=submit

3. Export the cartridge to a stand-alone html/js or native binary player (see the exporters section for details)

1.11 SPLORE

SPLORE is a built-in utility for browsing and organising both local and bbs (online) cartridges. Type SPLORE [enter] to launch it, or launch PICO-8 with -splore.

It is possible to control SPLORE entirely with a joystick:

```
LEFT and RIGHT to navigate lists of cartridges
UP AND DOWN to select items in each list
X,O or MENU to launch the cartridge
```

While inside a cart, press MENU to favourite a cartridge or exit to splore. If you're using a keyboard, it's also possible to press F to favourite an item while it is selected in the cartridge list view.

When viewing a list of BBS carts, use the top list item to re-download a list of cartridges. If you are offline, the last downloaded list is displayed, and it is still possible to play any cartridges you have downloaded.

If you have installed PICO-8 on a machine with no internet access, you can also use INSTALL_GAMES to add a small selection of pre-installed BBS carts to /games

❷ Editing Tools

Press ESC to toggle between console and editor.
Click editing mode tabs at top right to switch or press ALT+LEFT/RIGHT.

2.1 Code Editor

```
Hold shift to select (or click and drag with mouse)
CTRL-X, C, V to cut copy or paste selected
CTRL-Z, Y to undo, redo
CTRL-F to search for text in the current tab
CTRL-G to repeat the last search again
CTRL-L to jump to a line number
CTRL-UP, DOWN to jump to start or end
ALT-UP, DOWN to navigate to the previous, next function
CTRL-LEFT, RIGHT to jump by word
CTRL-W,E to jump to start or end of current line
CTRL-D to duplicate current line
TAB to indent a selection (shift to un-indent)
CTRL-B to comment / uncomment selected block
CTRL-U to get help on the keyword under the cursor
```

To enter special characters that represent buttons (and other glyphs), use SHIFT-L,R,U,D,O,X There are 3 additional font entry modes that can be toggled:

```
CTRL_J Hiragana □ // type romaji equivalents (ka, ki, ku..)
CTRL-K Katakana □ // + shift-0..9 for extra symbols
CTRL-P Puny font □// hold shift for the standard font
```

ⓘ By default, puny font characters are encoded as unicode replacements when copying/pasting, and both upper and lower case ASCII characters are pasted as regular PICO-8 characters. To copy/paste puny characters as uppercase ASCII, make sure puny mode (CTRL-P) is on.

□ Code Tabs

Click the [+] button at the top to add a new tab. Navigate tabs by left-clicking, or with CTRL-TAB, SHIFT-CTRL-TAB. To remove the last tab, delete any contents and then moving off it (CTRL-A, DEL, CTRL-TAB)

When running a cart, a single program is generated by concatenating all tabs in order.

□ Code Limits

The number of code tokens is shown at the bottom right. One program can have a maximum of 8192 tokens. Each token is a word (e.g. variable name) or operator. Pairs of brackets, and strings each count as 1 token. commas, periods, LOCALs, semi-colons, ENDs, and comments are not counted.

Right click to toggle through other stats (character count, compressed size). If a limit is reached, a warning light will flash. This can be disabled by right-clicking.

2.2 Sprite Editor

The sprite editor is designed to be used both for sprite-wise editing and for freeform pixel-level editing. The sprite navigator at the bottom of the screen provides an 8x8 sprite-wise view into the sprite sheet, but it is possible to use freeform tools (pan, select) when dealing with larger or oddly sized areas.

Draw Tool

Click and drag on the sprite to plot pixels, or use RMB to select the colour under the cursor.

All operations apply only to the visible area, or the section if there is one.

Hold CTRL to search and replace a colour.

Stamp Tool

Click to stamp whatever is in the copy buffer. Hold CTRL to treat colour 0 (black) as transparent.

Select Tool (shortcut: SHIFT or S)

Click and drag to create a rectangular selection. To remove the selection, press ENTER or click anywhere.

If a pixel-wise selection is not present, many operations are instead applied to a sprite-wise selection, or the visible view. To select sprites, shift-drag in the sprite navigator. To select the sprite sheet press CTRL-A (repeat to toggle off the bottom half shared with map data)

Pan Tool (shortcut: SPACE)

Click and drag to move around the sprite sheet.

Fill Tool

Fill with the current colour. This applies only to the current selection, or the visible area if there is no selection.

□ Shape Tools

Click the tool button to cycle through: oval, rectangle, line options.

Hold CTRL to get a filled oval or rectangle.
Hold SHIFT to snap to circle, square, or low-integer-ratio line.

□ Extra keys

CTRL-Z: □ Undo
CTRL-C/X: Copy/Cut selected area or selected sprites
CTRL-V: □ Paste to current sprite location
Q/A,W/Z: □Switch to previous/next sprite
1,2: □ □ □Switch to previous/next colour
TAB: □ □ □Toggle fullscreen view
Mousewheel or < and > to zoom (centered in fullscreen)
CTRL-H to toggle hex view (shows sprite index in hexadecimal)
CTRL-G to toggle black grid lines when zoomed in

□ Operations on selected area or selected sprites

F: Flip sprite horizontally
V: Flip sprite vertically
R: Rotate (requires a square selection)
Cursor keys to shift (loops if sprite selection)
DEL/BACKSPACE to clear selected area

□ Sprite Flags

The 8 coloured circles are sprite flags for the current sprite. These have no particular meaning, but can be accessed using the **FGET**() / **FSET**() functions. They are indexed from 0 starting from the left.

See **FSET**() for more information.

□ Loading .png files into the sprite sheet

To load a png file of any size into the sprite sheet, first select the sprite that should be the top-left corner destination, and then either type "IMPORT IMAGE_FILE.PNG" or drag and drop the image file into the PICO-8 window. In both cases, the image is colour-fitted to the current display palette.

2.3 Map Editor

The PICO-8 map is a 128x32 (or 128x64 using shared space) block of 8-bit values. Each value is shown in the editor as a reference to a sprite (0..255), but you can of course use the data to represent whatever you like.

ⓘ WARNING: The second half of the sprite sheet (banks 2 and 3), and the bottom half of the map share the same cartridge space. It's up to you how you use the data, but be aware that drawing on the second half of the sprite sheet could clobber data on the map and vice versa.

The tools are similar to the ones used in sprite editing mode. Select a sprite and click and drag to paint values into the map.

To draw multiple sprites, select from sprite navigator with shift+drag To copy a block of values, use the selection tool and then stamp tool to paste To pan around the map, use the pan tool or hold space Q,W to switch to previous/next sprite Mousewheel or < and > to zoom (centered in fullscreen) CTRL-H to toggle hex view (shows tile values and sprite index in hexadecimal)

Moving sprites in the sprite sheet without breaking reference to them in the map is a little tricky, but possible:

1. Press ENTER to clear any map selection
□ □ □ □ 2. Select the sprites you would like to move (while still in map view), and press Ctrl-X
□ □ □ □ 3. Select the area of the map you would like to alter (defaults to the top half of the map)
□ □ □ □ □ □ // press ctrl-A twice to select the full map including shared memory
□ □ □ □ 4. Select the destination sprite (also while still in map view) and press Ctrl-V

// Note: this operation modifies the undo history for both the map and sprite editors, but
// PICO-8 will try to keep them in sync where possible. Otherwise, changes caused by moving
// map sprites can be reverted by also manually undoing in the sprite editor.

2.4 SFX Editor

There are 64 SFX ("sound effects") in a cartridge, used for both sound and music.

Each SFX has 32 notes, and each note has:
□ A frequency □ (C0..C5)
□ An instrument (0..7)
□ A volume □ □ □(0..7)
□ An effect □ □ (0..7)

Each SFX also has these properties:

 A play speed (SPD) : the number of 'ticks' to play each note for.
  // This means that 1 is fastest, 3 is 3x as slow, etc.

 Loop start and end : this is the note index to loop back and to
  // Looping is turned off when the start index >= end index

When only the first of the 2 numbers is used (and the second one is 0), it is taken to mean the number of notes to be played. This is normally not needed for sound effects (you can just leave the remaining notes empty), but is useful for controlling music playback.

There are 2 modes for editing/viewing a SFX: Pitch mode (more suitable for sound effects) and tracker mode (more suitable for music). The mode can be changed using the top-left buttons, or toggled with TAB.

Pitch Mode

Click and drag on the pitch area to set the frequency for each note, using the currently selected instrument (indicated by colour).

Hold shift to apply only the selected instrument.
Hold CTRL to snap entered notes to the C minor pentatonic scale.
Right click to grab the instrument of that note.

Tracker Mode

Each note shows: frequency octave instrument volume effect
To enter a note, use q2w3er5t6y7ui zsxdcvgbhnjm (piano-like layout)
Hold shift when entering a note to transpose -1 octave .. +1 octave
New notes are given the selected instrument/effect values
To delete a note, use backspace or set the volume to 0

Click and then shift-click to select a range that can be copied (CTRL-C) and pasted (CTRL-V). Note that only the selected attributes are copied. Double-click to select all attributes of a single note.

Navigation:
 PAGEUP/DOWN or CTRL-UP/DOWN to skip up or down 4 notes
 HOME/END to jump to the first or last note
 CTRL-LEFT/RIGHT to jump across columns

Controls for both modes

- + to navigate the current SFX
< > to change the speed. SPACE to play/stop
SHIFT-SPACE to play from the current SFX quarter (group of 8 notes)
A to release a looping sample
Left click or right click to increase / decrease the SPD or LOOP values
 // Hold shift when clicking to increase / decrease by 4
 // Alternatively, click and drag left/right or up/down
Shift-click an instrument, effect, or volume to apply to all notes.

Effects

0 none
1 slide // Slide to the next note and volume
2 vibrato // Rapidly vary the pitch within one quarter-tone
3 drop // Rapidly drop the frequency to very low values
4 fade in // Ramp the volume up from 0
5 fade out // Ramp the volume down to 0
6 arpeggio fast // Iterate over groups of 4 notes at speed of 4
7 arpeggio slow // Iterate over groups of 4 notes at speed of 8

If the SFX speed is <= 8, arpeggio speeds are halved to 2, 4

Filters

Each SFX has 5 filter switches that can be accessed while in tracker mode:

NOIZ: Generate pure white noise (applies only to instrument 6)
BUZZ: Various alterations to the waveform to make it sound more buzzy
DETUNE-1: Detunes a second voice to create a flange-like effect
DETUNE-2: Various second voice tunings, mostly up or down an octave
REVERB: Apply an echo with a delay of 2 or 4 ticks
DAMPEN: Low pass filter at 2 different levels

When BUZZ is used with instrument 6, and NOIZ is off, pure brown noise is generated.

2.5 Music Editor

Music in PICO-8 is controlled by a sequence of 'patterns'. Each pattern is a list of 4 numbers indicating which SFX will be played on that channel.

Flow control

Playback flow can be controlled using the 3 buttons at the top right.

When a pattern has finished playing, the next pattern is played unless:

- there is no data left to play (music stops)
- a STOP command is set on that pattern (the third button)
- a LOOP BACK command is set (the 2nd button), in which case the music player searches back for a pattern with the LOOP START command set (the first button) or returns to pattern 0 if none is found.

When a pattern has SFXes with different speeds, the pattern finishes playing when the left-most non-looping channel has finished playing. This can be used to set up double-time drum beats or unusual polyrhythms.

For time signatures like 3/4 where less than 32 rows should be played before jumping to the next pattern, the length of a SFX can be set by adjusting only the first loop position and leaving the second one as zero. This will show up in the sfx editor as "LEN" (for "Length") instead of "LOOP".

Copying and Pasting Music

To select a range of patterns: click once on the first pattern in the pattern navigator, then shift-click on the last pattern. Selected patterns can be copied and pasted with CTRL-C and CTRL-V. When pasting into another cartridge, the SFX that each pattern points to will also be pasted (possibly with a different index) if it does not already exist.

SFX Instruments

In addition to the 8 built-in instruments, custom instruments can be defined using the first 8 SFX. Use the toggle button to the right of the instruments to select an index, which will show up in the instrument channel as green instead of pink.

When an SFX instrument note is played, it essentially triggers that SFX, but alters the note's attributes:

- Pitch is added relative to C2
- Volume is multiplied
- Effects are applied on top of the SFX instrument's effects
- Any filters that are on in the SFX instrument are enabled for that note

For example, a simple tremolo effect could be implemented by defining an instrument in SFX 0 that rapidly alternates between volume 5 and 2. When using this instrument to play a note, the volume can further be altered as usual (via the volume channel or using the fade in/out effects). In this way, SFX instruments can be used to control combinations of detailed changes in volume, pitch and texture.

SFX instruments are only retriggered when the pitch changes, or the previous note has zero volume. This is useful for instruments that change more slowly over time. For example: a bell that gradually fades out. To invert this behaviour, effect 3 (normally 'drop') can be used when triggering the note. All other effect values have their usual meaning when triggering SFX instruments.

□ Waveform Instruments

Waveform instruments function the same way as SFX instruments, but consist of a custom 64-byte looping waveform. Click on the waveform toggle button in the SFX editor to use SFX 0..7 as a waveform instrument. In this mode, samples can be drawn with the mouse.

□ Scale Snapping

When drawing notes in pitch mode, hold CTRL to snap to the currently defined scale. This is the C minor pentatonic scale by default, but can be customised using the scale editor mode. There is a little keyboard icon on the bottom right to toggle this. There are 2 tranpose buttons, 1 invert button, and 3 scale preset buttons:

```
Dim □6 diminished 6 scale - invert to get a whole-half scale
Maj □Major scale - invert to get pentatonic
Who □Whole tone scale - invert to get.. the other whole tone scale
```

Changing the scale does not alter the current SFX, it is only when drawing new notes with CTRL held down that the scale is applied.

❸ Exporters / Importers

The EXPORT command can be used to generate png, wav files and stand-alone html and native binary cartridge applications. The output format is inferred from the filename extension (e.g. .png).

You are free to distribute and use exported cartridges and data as you please, provided that you have permission from the cartridge author and contributors.

□ Sprite Sheet / Label (.png)

```
> IMPORT BLAH.PNG □ -- EXPECTS 128X128 PNG AND COLOUR-FITS TO THE PICO-8 PALETTE
> EXPORT BLAH.PNG □ -- USE THE "FOLDER" COMMAND TO LOCATE THE EXPORTED PNG
```

When importing, -x and -y switches can be used to specify the target location in pixels: -s can be used to shrink the image (3 means scale from 384x384 -> 128x128)

```
> IMPORT BLAH.PNG -X 16 -Y 16 -S 3
```

Use the -l switch with IMPORT and EXPORT to instead read and write from the cartridge's label:

```
> IMPORT -L BLAH.PNG
```

ⓘ When importing spritesheets or labels, the palette is colour-fitted to the current draw state palette.

SFX and Music (.wav)

To export music from the current pattern (when editor mode is MUSIC), or the current SFX:

```
> EXPORT FOO.WAV
```

To export all SFXs as foo0.wav, foo1.wav .. foo63.wav:

```
> EXPORT FOO%D.WAV
```

MAP and CODE

A cartridges map or source code can be exported as a single image named .map.png or .lua.png:

```
> EXPORT FOO.MAP.PNG
> EXPORT FOO.LUA.PNG
```

Map images are 1024x512 (128x32 8x8 sprites). Lua images are sized to fit, but each line is fixed (and cropped) at 192 pixels wide.

Cartridges (.p8, .p8.png, .p8.rom)

Using EXPORT to save a cartridge is the same as using SAVE, but without changing the current working cartridge. This can be useful for example, to save a copy in .p8.png format for distribution without accidentally continuing to make changes to that file instead of the original .p8 file.

EXPORT can also be used to perform cartridge file format conversions from commandline. For example, from a Linux shell:

```
> pico8 foo.p8 -export foo.p8.png
```

3.1 Web Applications (.html)

To generate a stand-alone html player (mygame.html, mygame.js):

```
> EXPORT MYGAME.HTML
```

Or just the .js file:

```
> EXPORT MYGAME.JS
```

Use -f to write the files to a folder called mygame_html, using index.html instead of mygame.html

```
> EXPORT -F MYGAME.HTML
```

Optionally provide a custom html template with the -p switch:

```
> EXPORT MYGAME.HTML -P ONE_BUTTON
```

This will use the file {application data}/pico-8/plates/one_button.html as the html shell, replacing a special string "##js_file##" (without quotes), with the .js filename, and optionally replacing the string "##label_file##" with the cart's label image as a data url.

Use -w to export as .wasm + .js:

```
> EXPORT -W MYGAME.HTML
```

ⓘ When exported as .wasm, the page needs to be served by a webserver, rather than just opening it directly from the local file system in a browser. For most purposes, the default .js export is fine, but .wasm is slightly smaller and faster.

3.2 Binary Applications (.bin)

To generate stand-alone executables for Windows, Linux (64-bit), Mac and Raspberry Pi:

```
> EXPORT MYGAME.BIN
```

By default, the cartridge label is used as an icon with no transparency. To specify an icon from the sprite sheet, use -i and optionally -s and/or -c to control the size and transparency.

```
-I N  Icon index N with a default transparent colour of 0 (black).
-S N  Size NxN sprites. Size 3 would be produce a 24x24 icon.
-C N  Treat colour N as transparent. Use 16 for no transparency.
```

For example, to use a 2x2 sprite starting at index 32 in the sprite sheet, using colour 12 as transparent:

```
> EXPORT -I 32 -S 2 -C 12 MYGAME.BIN
```

To include an extra file in the output folders and archives, use the -E switch:

```
> EXPORT -E README.TXT MYGAME.BIN
```

ⓘ Windows file systems do not support the file metadata needed to create a Linux or Mac executable. PICO-8 works around this by exporting zip files in a way that preserves the file attributes. It is therefore recommended that you distribute the outputted zip files as-is to ensure users on other operating systems can run them. Otherwise, a Linux user who then downloads the binaries may need to "chmod +x mygame" the file to run it, and Mac user would need to "chmod +x mygame.app/Contents/MacOS/mygame"

3.3 Uploading to itch.io

If you would like to upload your exported cartridge to itch.io as playable html:

1. From inside PICO-8: EXPORT -F MYGAME.HTML
2. Create a new project from your itch dashboard.
3. Zip up the folder and upload it (set "This file will be played in the browser")
4. Embed in page, with a size of 750px x 680px.
5. Set "Mobile Friendly" on (default orientation) and "Automatically start on page load" on.
   // no need for the fullscreen button as the default PICO-8 template has its own.
6. Set the background (BG2) to something dark (e.g. #232323) and the text to something light (#cccccc)

3.4 Exporting Multiple Cartridges

Up to 16 cartridges can be bundled together by passing them to EXPORT, when generating stand-alone html or native binary players.

```
> EXPORT MYGAME.HTML DAT1.P8 DAT2.P8 GAME2.P8
```

During runtime, the extra carts can be accessed as if they were local files:

```
RELOAD(0,0,0X2000, "DAT1.P8") -- LOAD SPRITESHEET FROM DAT1.P8
LOAD("GAME2.P8")              -- LOAD AND RUN ANOTHER CART
```

ⓘ Exported cartridges are unable to load and run BBS cartridges e.g. via LOAD("#FOO")

3.5 Running EXPORT from the host operating system

Use the -export switch when launching PICO-8 to run the exporter in headless mode. File paths are relative to the current directory rather than the PICO-8 file system.

Parameters to the EXPORT command are passed along as a single (lowercase) string:

```
pico8 mygame.p8 -export "-i 32 -s 2 -c 12 mygame.bin dat0.p8 dat1.p8"
```

❹ Lua Syntax Primer

PICO-8 programs are written using Lua syntax, but do not use the standard Lua library. The following is a brief summary of essential Lua syntax.

For more details, or to find out about proper Lua, see www.lua.org.

□ Comments

```
-- USE TWO DASHES LIKE THIS TO WRITE A COMMENT
--[[ MULTI-LINE
COMMENTS ]]
```

□ Types and assignment

Types in Lua are numbers, strings, booleans and tables:

```
NUM = 12/100
S = "THIS IS A STRING"
B = FALSE
T = {1,2,3}
```

Numbers in PICO-8 are all 16:16 fixed point. They range from -32768.0 to 32767.99999

Hexadecimal notation with optional fractional parts can be used:

```
?0x11 □ □ □ □-- 17
?0x11.4000 □ -- 17.25
```

Numbers written in decimal are rounded to the closest fixed point value. To see the 32-bit hexadecimal representation, use PRINT(TOSTR(VAL,TRUE)):

```
?TOSTR(-32768,TRUE) □ □ □-- 0x8000.0000
?TOSTR(32767.99999,TRUE) -- 0X7FFF.FFFF
```

Dividing by zero evaluates to 0x7fff.ffff if positive, or -0x7fff.ffff if negative.

Conditionals

```
IF NOT B THEN
 PRINT("B IS FALSE")
ELSE
 PRINT("B IS NOT FALSE")
END
```

-- with ELSEIF

```
IF X == 0 THEN
 PRINT("X IS 0")
ELSEIF X < 0 THEN
 PRINT("X IS NEGATIVE")
ELSE
 PRINT("X IS POSITIVE")
END
```

```
IF (4 == 4) THEN PRINT("EQUAL") END
IF (4 ~= 3) THEN PRINT("NOT EQUAL") END
IF (4 <= 4) THEN PRINT("LESS THAN OR EQUAL") END
IF (4 > 3) THEN PRINT("MORE THAN") END
```

Loops

Loop ranges are inclusive:

```
FOR X=1,5 DO
 PRINT(X)
END
-- PRINTS 1,2,3,4,5
```

```
X = 1
WHILE(X <= 5) DO
 PRINT(X)
 X = X + 1
END
```

```
FOR X=1,10,3 DO PRINT(X) END  -- 1,4,7,10
```

```
FOR X=5,1,-2 DO PRINT(X) END --5,3,1
```

Functions and Local Variables

Variables declared as LOCAL are scoped to their containing block of code (for example, inside a FUNCTION, a FOR loop, or IF THEN END statement).

```
Y=0
FUNCTION PLUSONE(X)
 LOCAL Y = X+1
 RETURN Y
END
PRINT(PLUSONE(2)) -- 3
PRINT(Y)  -- 0
```

Tables

In Lua, tables are a collection of key-value pairs where the key and value types can both be mixed. They can be used as arrays by indexing them with integers.

```
A={} -- CREATE AN EMPTY TABLE
A[1] = "BLAH"
A[2] = 42
A["FOO"] = {1,2,3}
```

Arrays use 1-based indexing by default:

```
> A = {11,12,13,14}
> PRINT(A[2]) -- 12
```

But if you prefer 0-based arrays, just write something the zeroth slot:

```
> A = {[0]=10,11,12,13,14}
```

Tables with 1-based integer indexes are special though. The length of such an array can be found with the # operator, and PICO-8 uses such arrays to implement ADD, DEL, DELI, ALL and FOREACH functions.

```
> PRINT(#A) □ -- 4
> ADD(A, 15)
> PRINT(#A) □ -- 5
```

Indexes that are strings can be written using dot notation

```
PLAYER = {}
PLAYER.X = 2 -- is equivalent to PLAYER["X"]
PLAYER.Y = 3
```

See the **Table_Functions** section for more details.

□ PICO-8 Shorthand

PICO-8 also allows several non-standard, shorter ways to write common patterns.

1. IF THEN END statements, and WHILE THEN END can be written on a single line with:

```
IF (NOT B) I=1 J=2
```

Is equivalent to:

```
IF NOT B THEN I=1 J=2 END
```

Note that brackets around the short-hand condition are required.

2. Assignment operators

Shorthand assignment operators can also be used if the whole statement is on one line. They can be constructed by appending a '=' to any binary operator, including arithmetic (+=, -= ..), bitwise (&=, |= ..) or the string concatenation operator (..=)

```
A += 2 □ -- EQUIVALENT TO: A = A + 2
```

// note that the LHS appears twice, so for TBL[FN()]+=1, FN() will be called twice.

3. != operator

Not shorthand, but pico-8 also accepts != instead of ~= for "not equal to"

```
PRINT(1 != 2) -- TRUE
PRINT("FOO" == "FOO") -- TRUE (STRING ARE INTERNED)
```

❺ PICO-8 Program Structure

When a PICO-8 programs runs, all of the code from tabs is concatenated (from left to right) and executed. It is possible to provide your own main loop manually, but typically PICO-8 programs use 3 special functions that, if defined by the author, are called during program execution:

_UPDATE() -- Called once per update at 30fps.

_DRAW() -- Called once per visible frame

_INIT() -- Called once on program startup.

A simple program that uses all three might look this:

```
FUNCTION _INIT()
 -- ALWAYS START ON WHITE
 COL = 7
END

FUNCTION _UPDATE()
 -- PRESS X FOR A RANDOM COLOUR
 IF (BTNP(5)) COL = 8 + RND(8)
END

FUNCTION _DRAW()
 CLS(1)
 CIRCFILL(64,64,32,COL)
END
```

_DRAW() is normally called at 30fps, but if it can not complete in time, PICO-8 will attempt to run at 15fps and call _UPDATE() twice per visible frame to compensate.

Running PICO-8 at 60fps

_UPDATE60()

When _UPDATE60() Is defined instead of _UPDATE(), PICO-8 will run in 60fps mode:

- both _UPDATE60() and _DRAW() are called at 60fps
- half the PICO-8 CPU is available per frame before dropping down to 30fps

Note that not all host machines are capable of running at 60fps. Older machines, and / or web versions might also request PICO-8 to run at 30 fps (or 15 fps), even when the PICO-8 CPU is not over capacity. In this case, multiple _UPDATE60 calls are made for every _DRAW call in the same way.

#INCLUDE

Source code can be injected into a program at cartridge boot (but not during runtime), using "#INCLUDE FILENAME", where FILENAME is either a plaintext file (containing Lua code), a tab from another cartridge, or all tabs from another cartridge:

```
#INCLUDE SOMECODE.LUA
#INCLUDE ONETAB.P8:1
#INCLUDE ALLTABS.P8
```

When the cartridge is run, the contents of each included file is treated as if it had been pasted into the editor in place of that line.

- Filenames are relative to the current cartridge (so, need to save first)
- Includes are not performed recursively.
- Normal character count and token limits apply.

When a cartridge is saved as .P8.PNG, or exported to a binary, any included files are flattened and saved with the cartridge so that there are no external dependencies.

#INCLUDE can be used for things like:

- Sharing code between cartridge (libraries or common multi-cart code)
- Using an external code editor without needing to edit the .p8 file directly.
- Treating a cartridge as a data file that loads a PICO-8 editing tool to modify it.
- Loading and storing data generated by an external (non-PICO-8) tool.

□ Quirks of PICO-8

Common gotchas to watch out for:

- The bottom half of the sprite sheet and bottom half of the map occupy the same memory.
// Best use only one or the other if you're unsure how this works.

- PICO-8 numbers have limited accuracy and range; the minimum step between numbers is approximately 0.00002 (0x0.0001), with a range of -32768 (-0x8000) to approximately 32767.99999 (0x7fff.ffff)
// If you add 1 to a counter each frame, it will overflow after around 18 minutes!

- Lua arrays are 1-based by default, not 0-based. FOREACH starts at TBL[1], not TBL[0].

- **COS**() and **SIN**() take 0..1 instead of 0..PI*2, and **SIN**() is inverted.

- **SGN**(0) returns 1.

□ CPU

Although PICO-8 does not have a clearly defined CPU, there is a virtual CPU speed of 8MHz, where each lua vm instruction costs around 2 cycles. Built-in operations like drawing sprites also have a CPU cost. This means that a PICO-8 cartridge made on a host machine with a powerful CPU can still be guaranteed to run (reasonably) well on much slower machines, and to not drain too much battery on phones / when running on the web.

To view the CPU load while a cartridge is running, press CTRL-P to toggle a CPU meter, or print out **STAT**(1) at the end of each frame.

❻ API Reference

PICO-8 is built on the Lua programming language, but does not include the Lua standard library. Instead, a small api is offered in keeping with PICO-8's minimal design and limited screen space. For an example program that uses most of the api functions, see /DEMOS/API.P8

Functions are written here as:

FUNCTION_NAME(PARAMETER, [OPTIONAL_PARAMETER])

ⓘ Note that PICO-8 does not have upper or lower case characters -- if you are editing a .p8 or .lua file directly, function names should all be in lower case.

6.1 System

System functions called from commandline can omit the usual brackets and string quotes. For example, instead of LOAD("BLAH.P8"), it is possible to write:

```
>LOAD BLAH.P8
```

LOAD(FILENAME, [BREADCRUMB], [PARAM_STR])

SAVE(FILENAME)

Load or save a cartridge

When loading from a running cartridge, the loaded cartridge is immediately run with parameter string PARAM_STR (accessible with STAT(6)), and a menu item is inserted and named BREADCRUMB, that returns the user to the previous cartridge.

Filenames that start with '#' are taken to be a BBS cart id, that is immediately downloaded and run:

```
> LOAD("#MYGAME_LEVEL2", "BACK TO MAP", "LIVES="..LIVES)
```

If the id is the cart's parent post, or a revision number is not specified, then the latest version is fetched. BBS carts can be loaded from other BBS carts or local carts, but not from exported carts.

FOLDER

Open the carts folder in the host operating system.

LS([DIRECTORY])

List .p8 and .p8.png files in given directory (folder), relative to the current directory. Items that are directories end in a slash (e.g. "foo/").

When called from a running cartridge, LS can only be used locally and returns a table of the results. When called from a BBS cart, LS returns nil.

Directories can only resolve inside of PICO-8's virtual drive; LS("..") from the root directory will resolve to the root directory.

RUN([PARAM_STR])

Run from the start of the program.

RUN() Can be called from inside a running program to reset.

When PARAM_STR is supplied, it can be accessed during runtime with STAT(6)

STOP([MESSAGE])

Stop the cart and optionally print a message.

RESUME

Resume the program. Use R for short.

Use a single "." from the commandline to advance a single frame. This enters frame-by-frame mode, that can be read with stat(110). While frame-by-frame mode is active, entering an empty command (by pressing enter) advances one frames.

ASSERT(CONDITION, [MESSAGE])

If CONDITION is false, stop the program and print MESSAGE if it is given. This can be useful for debugging cartridges, by ASSERT()'ing that things that you expect to be true are indeed true.

```
ASSERT(ADDR >= 0 AND ADDR <= 0x7FFF, "OUT OF RANGE")
POKE(ADDR, 42) -- THE MEMORY ADDRESS IS OK, FOR SURE!
```

REBOOT

Reboot the machine Useful for starting a new project

RESET()

Reset the values in RAM from 0x5f00..0x5f7f to their default values. This includes the palette, camera position, clipping and fill pattern. If you get lost at the command prompt because the draw state makes viewing text impossible, try typing RESET! It can also be called from a running program.

INFO()

Print out some information about the cartridge: Code size, tokens, compressed size

Also displayed:

UNSAVED CHANGES □ When the cartridge in memory differs to the one on disk
EXTERNAL CHANGES □When the cartridge on disk has changed since it was loaded
□ (e.g. by editing the program using a separate text editor)

FLIP()

Flip the back buffer to screen and wait for next frame. This call is not needed when there is a **_DRAW**() or **_UPDATE**() callback defined, as the flip is performed automatically. But when using a custom main loop, a call to FLIP is normally needed:

```
::_::
CLS()
FOR I=1,100 DO
□ A=I/50 - T()
□ X=64+COS(A)*I
□ Y=64+SIN(A)*I
□ CIRCFILL(X,Y,1,8+(I/4)%8)
END
FLIP()GOTO _
```

If your program does not call FLIP before a frame is up, and a **_DRAW**() callback is not in progress, the current contents of the back buffer are copied to screen.

PRINTH(STR, [FILENAME], [OVERWRITE], [SAVE_TO_DESKTOP])

Print a string to the host operating system's console for debugging.

If filename is set, append the string to a file on the host operating system (in the current directory by default -- use FOLDER to view).

Setting OVERWRITE to true causes that file to be overwritten rather than appended.

Setting SAVE_TO_DESKTOP to true saves to the desktop instead of the current path.

Use a filename of "@clip" to write to the host's clipboard.

ⓘ Use stat(4) to read the clipboard, but the contents of the clipboard are only available after pressing CTRL-V during runtime (for security).

TIME()

T()

Returns the number of seconds elapsed since the cartridge was run.

This is not the real-world time, but is calculated by counting the number of times _UPDATE or @_UPDATE60 is called. Multiple calls of TIME() from the same frame return the same result.

STAT(X)

Get system status where X is:

```
0 □Memory usage (0..2048)
1 □CPU used since last flip (1.0 == 100% CPU)
4 □Clipboard contents (after user has pressed CTRL-V)
6 □Parameter string
7 □Current framerate

46..49 □Index of currently playing SFX on channels 0..3
50..53 □Note number (0..31) on channel 0..3
54 □ □ □Currently playing pattern index
55 □ □ □Total patterns played
56 □ □ □Ticks played on current pattern
57 □ □ □(Boolean) TRUE when music is playing

80..85 □UTC time: year, month, day, hour, minute, second
90..95 □Local time

100 □ □ Current breadcrumb label, or nil
110 □ □ Returns true when in frame-by-frame mode
```

ⓘ Audio values 16..26 are the legacy version of audio state queries 46..56. They only report on the current state of the audio mixer, which changes only ~20 times a second (depending on the host sound driver and other factors). 46..56 instead stores a history of mixer state at each tick to give a higher resolution estimate of the currently audible state.

EXTCMD(CMD_STR, [P1, P2])

Special system command, where CMD_STR is a string:

```
"pause" □ □ □ □ request the pause menu be opened
"reset" □ □ □ □ request a cart reset
"go_back" □ □ □ return to the previous cart if there is one
"label" □ □ □ □ set cart label
"screen" □ □ □ □save a screenshot
"rec" □ □ □ □ □ set video start point
"rec_frames" □ □set video start point in frames mode
"video" □ □ □ □ save a .gif to desktop
"audio_rec" □ □ start recording audio
"audio_end" □ □ save recorded audio to desktop (no supported from web)
"shutdown" □ □ □quit cartridge (from exported binary)
"folder" □ □ □ □open current working folder on the host operating system
"set_filename" □set the filename for screenshots / gifs / audio recordings
"set_title" □ □ set the host window title
```

Some commands have optional number parameters:

"video" and "screen": P1: an integer scaling factor that overrides the system setting. P2: when > 0, save to the current folder instead of to desktop

"audio_end" P1: when > 0, save to the current folder instead of to desktop

Recording GIFs

EXTCMD("REC"), EXTCMD("VIDEO") is the same as using ctrl-8, ctrl-9 and saves a gif to the desktop using the current GIF_SCALE setting (use CONFIG GIF_SCALE to change).

The two additional parameters can be used to override these defaults:

```
EXTCMD("VIDEO", 4)  -- SCALE *4 (512 X 512)
EXTCMD("VIDEO", 0, 1) -- DEFAULT SCALING, SAVE TO USER DATA FOLDER
```

The user data folder can be opened with EXTCMD("FOLDER") and defaults to the same path as the cartridge, or {pico-8 appdata}/appdata/appname for exported binaries.

Due to the nature of the gif format, all gifs are recorded at 33.3fps, and frames produced by PICO-8 are skipped or duplicated in the gif to match roughly what the user is seeing. To record exactly one frame each time **FLIP**() is called, regardless of the runtime framerate or how long it took to generate the frame, use:

```
EXTCMD("REC_FRAMES")
```

The default filename for gifs (and screenshots, audio) is foo_%d, where foo is the name of the cartridge, and %d is a number starting at 0 and automatically incremented until a file of that name does not exist. Use EXTCMD("SET_FILENAME","FOO") to override that default. If the custom filename includes "%d", then the auto- incrementing number behaviour is used, but otherwise files are written even if there is an existing file with the same name.

6.2 Graphics

PICO-8 has a fixed capacity of 128 8x8 sprites, plus another 128 that overlap with the bottom half of the map data ("shared data"). These 256 sprites are collectively called the sprite sheet, and can be thought of as a 128x128 pixel image.

All of PICO-8's drawing operations are subject to the current draw state. The draw state includes a camera position (for adding an offset to all coordinates), palette mapping (for recolouring sprites), clipping rectangle, a drawing colour, and a fill pattern.

The draw state is reset each time a program is run, or by calling **RESET**().

Colour indexes:

```
□ □0 □black □ 1 □dark_blue □ 2 □dark_purple □ 3 □dark_green □
□ □4 □brown □ 5 □dark_gray □ 6 □light_gray □ □7 □white
□ □8 □red □ □ 9 □orange □ □ 10 □yellow □ □ □ 11 □green □ □ □
□ 12 □blue □ 13 □indigo □ □ 14 □pink □ □ □ □ 15 □peach
```

CLIP(X, Y, W, H, [CLIP_PREVIOUS])

Sets the clipping rectangle in pixels. All drawing operations will be clipped to the rectangle at x, y with a width and height of w,h.

CLIP() to reset.

When CLIP_PREVIOUS is true, clip the new clipping region by the old one.

PSET(X, Y, [COL])

Sets the pixel at x, y to colour index COL (0..15).

When COL is not specified, the current draw colour is used.

```
FOR Y=0,127 DO
□ FOR X=0,127 DO
□ □ PSET(X, Y, X*Y/8)
□ END
END
```

PGET(X, Y)

Returns the colour of a pixel on the screen at (X, Y).

```
WHILE (TRUE) DO
□ X, Y = RND(128), RND(128)
□ DX, DY = RND(4)-2, RND(4)-2
□ PSET(X, Y, PGET(DX+X, DY+Y))
END
```

When X and Y are out of bounds, PGET returns 0. A custom return value can be specified with:

```
POKE(0x5f36, 0x10)
POKE(0x5f5B, NEWVAL)
```

SGET(X, Y)

SSET(X, Y, [COL])

Get or set the colour (COL) of a sprite sheet pixel.

When X and Y are out of bounds, SGET returns 0. A custom value can be specified with:

```
POKE(0x5f36, 0x10)
POKE(0x5f59, NEWVAL)
```

FGET(N, [F])

FSET(N, [F], VAL)

Get or set the value (VAL) of sprite N's flag F.

F is the flag index 0..7.

VAL is TRUE or FALSE.

The initial state of flags 0..7 are settable in the sprite editor, so can be used to create custom sprite attributes. It is also possible to draw only a subset of map tiles by providing a mask in **MAP**().

When F is omitted, all flags are retrieved/set as a single bitfield.

```
FSET(2, 1 | 2 | 8) □ -- SETS BITS 0,1 AND 3
FSET(2, 4, TRUE) □ □ -- SETS BIT 4
PRINT(FGET(2)) □ □ □ -- 27 (1 | 2 | 8 | 16)
```

PRINT(STR, X, Y, [COL])

PRINT(STR, [COL])

Print a string STR and optionally set the draw colour to COL.

Shortcut: written on a single line, ? can be used to call print without brackets:

```
?"HI"
```

When X, Y are not specified, a newline is automatically appended. This can be omitted by ending the string with an explicit termination control character:

```
?"THE QUICK BROWN FOX\0"
```

Additionally, when X, Y are not specified, printing text below 122 causes the console to scroll. This can be disabled during runtime with POKE(0x5f36,0x40).

PRINT returns the right-most x position that occurred while printing. This can be used to find out the width of some text by printing it off-screen:

```
W = PRINT("HOGE", 0, -20) -- returns 16
```

See **Appendix A** (P8SCII) for information about control codes and custom fonts.

CURSOR(X, Y, [COL])

Set the cursor position.

If COL is specified, also set the current colour.

COLOR([COL])

Set the current colour to be used by drawing functions.

If COL is not specified, the current colour is set to 6

CLS([COL])

Clear the screen and reset the clipping rectangle.

COL defaults to 0 (black)

CAMERA([X, Y])

Set a screen offset of -x, -y for all drawing operations

CAMERA() to reset

CIRC(X, Y, R, [COL])

CIRCFILL(X, Y, R, [COL])

Draw a circle or filled circle at x,y with radius r

If r is negative, the circle is not drawn.

When bits 0x1800.0000 are set in COL, and **0x5F34** & 2 == 2, the circle is drawn inverted.

OVAL(X0, Y0, X1, Y1, [COL])

OVALFILL(X0, Y0, X1, Y1, [COL])

Draw an oval that is symmetrical in x and y (an ellipse), with the given bounding rectangle.

LINE(X0, Y0, [X1, Y1, [COL]])

Draw a line from (X0, Y0) to (X1, Y1)

If (X1, Y1) are not given, the end of the last drawn line is used.

LINE() with no parameters means that the next call to LINE(X1, Y1) will only set the end points without drawing.

```
CLS()
LINE()
FOR I=0,6 DO
 LINE(64+COS(I/6)*20, 64+SIN(I/6)*20, 8+I)
END
```

RECT(X0, Y0, X1, Y1, [COL])

RECTFILL(X0, Y0, X1, Y1, [COL])

Draw a rectangle or filled rectangle with corners at (X0, Y0), (X1, Y1).

PAL(C0, C1, [P])

PAL() swaps colour c0 for c1 for one of three palette re-mappings (p defaults to 0):

0: Draw Palette

The draw palette re-maps colours when they are drawn. For example, an orange flower sprite can be drawn as a red flower by setting the 9th palette value to 8:

```
PAL(9,8) □ □ -- draw subsequent orange (colour 9) pixels as red (colour 8)
SPR(1,70,60) -- any orange pixels in the sprite will be drawn with red instead
```

Changing the draw palette does not affect anything that was already drawn to the screen.

1: Display Palette

The display palette re-maps the whole screen when it is displayed at the end of a frame. For example, if you boot PICO-8 and then type PAL(6,14,1), you can see all of the gray (colour 6) text immediate change to pink (colour 14) even though it has already been drawn. This is useful for screen-wide effects such as fading in/out.

2: Secondary Palette

Used by **FILLP**() for drawing sprites. This provides a mapping from a single 4-bit colour index to two 4-bit colour indexes.

PAL() □resets all palettes to system defaults (including transparency values)
PAL(P) resets a particular palette (0..2) to system defaults

PAL(TBL, [P])

When the first parameter of pal is a table, colours are assigned for each entry. For example, to re-map colour 12 and 14 to red:

```
PAL({[12]=9, [14]=8})
```

Or to re-colour the whole screen shades of gray (including everything that is already drawn):

```
PAL({1,1,5,5,5,6,7,13,6,7,7,6,13,6,7,1}, 1)
```

Because table indexes start at 1, colour 0 is given at the end in this case.

PALT(C, [T])

Set transparency for colour index to T (boolean) Transparency is observed by **SPR**(), **SSPR**(), **MAP**() AND **TLINE**()

```
PALT(8, TRUE) -- RED PIXELS NOT DRAWN IN SUBSEQUENT SPRITE/TLINE DRAW CALLS
```

PALT() resets to default: all colours opaque except colour 0

When C is the only parameter, it is treated as a bitfield used to set all 16 values. For example: to set colours 0 and 1 as transparent:

```
PALT(0B1100000000000000)
```

SPR(N, X, Y, [W, H], [FLIP_X], [FLIP_Y])

Draw sprite N (0..255) at position X,Y

W (width) and H (height) are 1, 1 by default and specify how many sprites wide to blit.

Colour 0 drawn as transparent by default (see **PALT**())

When FLIP_X is TRUE, flip horizontally.

When FLIP_Y is TRUE, flip vertically.

SSPR(SX, SY, SW, SH, DX, DY, [DW, DH], [FLIP_X], [FLIP_Y]]

Stretch an rectangle of the sprite sheet (sx, sy, sw, sh) to a destination rectangle on the screen (sx, sy, dw, dh). In both cases, the x and y values are coordinates (in pixels) of the rectangle's top left corner, with a width of w, h.

Colour 0 drawn as transparent by default (see **PALT**())

dw, dh defaults to sw, sh

When FLIP_X is TRUE, flip horizontally.

When FLIP_Y is TRUE, flip vertically.

FILLP(P)

The PICO-8 fill pattern is a 4x4 2-colour tiled pattern observed by: **CIRC**() **CIRCFILL**() **RECT**() **RECTFILL**() **OVAL**() **OVALFILL**() **PSET**() **LINE**()

P is a bitfield in reading order starting from the highest bit. To calculate the value of P for a desired pattern, add the bit values together:

```
 .-----------------------.
 |32768|16384| 8192| 4096|
 |-----|-----|-----|-----|
 | 2048| 1024| 512 | 256 |
 |-----|-----|-----|-----|
 | 128 | 64  | 32  | 16  |
 |-----|-----|-----|-----|
 | 8   | 4   | 2   | 1   |
 '-----------------------'
```

For example, FILLP(4+8+64+128+ 256+512+4096+8192) would create a checkerboard pattern.

This can be more neatly expressed in binary: FILLP(0b0011001111001100).

The default fill pattern is 0, which means a single solid colour is drawn.

To specify a second colour for the pattern, use the high bits of any colour parameter:

```
FILLP(0b0011010101101000)
CIRCFILL(64,64,20, 0x4E) -- brown and pink
```

Additional settings are given in bits 0b0.111:

0b0.100 Transparency

When this bit is set, the second colour is not drawn

```
-- checkboard with transparent squares
FILLP(0b0011001111001100.1)
```

0b0.010 Apply to Sprites

When set, the fill pattern is applied to sprites (spr, sspr, map, tline), using a colour mapping provided by the secondary palette.

Each pixel value in the sprite (after applying the draw palette as usual) is taken to be an index into the secondary palette. Each entry in the secondary palette contains the two colours used to render the fill pattern. For example, to draw a white and red (7 and 8) checkerboard pattern for only blue pixels (colour 12) in a sprite:

```
FOR I=0,15 DO PAL(I, I+I*16, 2) END -- all other colours map to themselves
PAL(12, 0x87, 2) -- remap colour 12 in the secondary palette

FILLP(0b0011001111001100.01) -- checkerboard palette, applied to sprites
SPR(1, 64,64) -- draw the sprite
```

0b0.001 Apply Secondary Palette Globally

When set, the secondary palette mapping is also applied by all draw functions that respect fill patterns (circfill, line etc). This can be useful when used in conjunction with sprite drawing functions, so that the colour index of each sprite pixel means the same thing as the colour index supplied to the drawing functions.

```
FILLP(0b0011001111001100.001)
PAL(12, 0x87, 2)
CIRCFILL(64,64,20,12) □ □ □ □ □ □ □ □-- red and white checkerboard circle
```

The secondary palette mapping is applied after the regular draw palette mapping. So the following would also draw a red and white checkered circle:

```
PAL(3,12)
CIRCFILL(64,64,20,3)
```

The fill pattern can also be set by setting bits in any colour parameter (for example, the parameter to **COLOR**(), or the last parameter to **LINE**(), **RECT**() etc.

```
POKE(0x5F34, 0x3) -- 0x1 enable fillpattern in high bits □0x2 enable inversion mode
CIRCFILL(64,64,20, 0x114E.ABCD) -- sets fill pattern to ABCD
```

When using the colour parameter to set the fill pattern, the following bits are used:

```
bit □0x1000.0000 this needs to be set: it means "observe bits 0xf00.ffff"
bit □0x0100.0000 transparency
bit □0x0200.0000 apply to sprites
bit □0x0400.0000 apply secondary palette globally
bit □0x0800.0000 invert the drawing operation (circfill/ovalfill/rectfill)
bits 0x00FF.0000 are the usual colour bits
bits 0x0000.FFFF are interpreted as the fill pattern
```

6.3 Table Functions

With the exception of PAIRS(), the following functions and the # operator apply only to tables that are indexed starting from 1 and do not have NIL entries. All other forms of tables can be considered as hash maps or sets, rather than arrays that have a length.

ADD(TBL, VAL, [INDEX])

Add value VAL to the end of table TBL. Equivalent to:

```
TBL[#TBL + 1] = VAL
```

If index is given then the element is inserted at that position:

```
FOO={} □ □ □ □-- CREATE EMPTY TABLE
ADD(FOO, 11)
ADD(FOO, 22)
PRINT(FOO[2]) -- 22
```

DEL(TBL, VAL)

Delete the first instance of value VAL in table TBL. The remaining entries are shifted left one index to avoid holes.

Note that VAL is the value of the item to be deleted, not the index into the table. (To remove an item at a particular index, use DELI instead). DEL returns the deleted item, or returns no value when nothing was deleted.

```
A={1,10,2,11,3,12}
FOR ITEM IN ALL(A) DO
□ IF (ITEM < 10) THEN DEL(A, ITEM) END
END
FOREACH(A, PRINT) -- 10,11,12
PRINT(A[3]) □ □ □ -- 12
```

DELI(TBL, [I])

Like **DEL**(), but remove the item from table TBL at index I When I is not given, the last element of the table is removed and returned.

COUNT(TBL, [VAL])

Returns the length of table t (same as #TBL) When VAL is given, returns the number of instances of VAL in that table.

ALL(TBL)

Used in FOR loops to iterate over all items in a table (that have a 1-based integer index), in the order they were added.

```
T = {11,12,13}
ADD(T,14)
ADD(T,"HI")
FOR V IN ALL(T) DO PRINT(V) END -- 11 12 13 14 HI
PRINT(#T) -- 5
```

FOREACH(TBL, FUNC)

For each item in table TBL, call function FUNC with the item as a single parameter.

```
> FOREACH({1,2,3}, PRINT)
```

PAIRS(TBL)

Used in FOR loops to iterate over table TBL, providing both the key and value for each item. Unlike **ALL**(), PAIRS() iterates over every item regardless of indexing scheme. Order is not guaranteed.

```
T = {["HELLO"]=3, [10]="BLAH"}
T.BLUE = 5;
FOR K,V IN PAIRS(T) DO
 PRINT("K: "..K.." V:"..V)
END
```

Output:

```
K: 10 v:BLAH
K: HELLO v:3
K: BLUE v:5
```

6.4 Input

BTN([B], [PL])

Get button B state for player PL (default 0)

B: 0..5: left right up down button_o button_x
PL: player index 0..7

Instead of using a number for B, it is also possible to use a button glyph. (In the coded editor, use Shift-L R U D O X)

If no parameters supplied, returns a bitfield of all 12 button states for player 0 & 1
// P0: bits 0..5 P1: bits 8..13

Default keyboard mappings to player buttons:

□ player 0: [DPAD]: cursors, [O]: Z C N □ [X]: X V M
□ player 1: [DPAD]: SFED, □ □[O]: LSHIFT □[X]: TAB W □Q A

Although PICO-8 accepts all button combinations, note that it is generally impossible to press both LEFT and RIGHT at the same time on a physical game controller. On some controllers, UP + LEFT/RIGHT is also awkward if [X] or [O] could be used instead of UP (e.g. to jump / accelerate).

BTNP(B, [PL])

BTNP is short for "Button Pressed"; Instead of being true when a button is held down, BTNP returns true when a button is down AND it was not down the last frame. It also repeats after 15 frames, returning true every 4 frames after that (at 30fps -- double that at 60fps). This can be used for things like menu navigation or grid-wise player movement.

The state that BTNP reads is reset at the start of each call to **_UPDATE** or **_UPDATE60**, so it is preferable to use BTNP from inside one of those functions.

Custom delays (in frames 30fps) can be set by poking the following memory addresses:

```
POKE(0X5F5C, DELAY) -- SET THE INITIAL DELAY BEFORE REPEATING. 255 MEANS NEVER
REPEAT.
POKE(0X5F5D, DELAY) -- SET THE REPEATING DELAY.
```

In both cases, 0 can be used for the default behaviour (delays 15 and 4)

6.5 Audio

SFX(N, [CHANNEL], [OFFSET], [LENGTH])

Play sfx N (0..63) on CHANNEL (0..3) from note OFFSET (0..31 in notes) for LENGTH notes.

Using negative CHANNEL values have special meanings:

CHANNEL -1: (default) to automatically choose a channel that is not being used
CHANNEL -2: to stop the given sound from playing on any channel

N can be a command for the given CHANNEL (or all channels when CHANNEL < 0):

N -1: to stop sound on that channel
N -2: to release sound on that channel from looping

```
SFX(3)  --  PLAY SFX 3
SFX(3,2) --  PLAY SFX 3 ON CHANNEL 2
SFX(3,-2) --  STOP SFX 3 FROM PLAYING ON ANY CHANNEL
SFX(-1,2) --  STOP WHATEVER IS PLAYING ON CHANNEL 2
SFX(-2,2) --  RELEASE LOOPING ON CHANNEL 2
SFX(-1)  --  STOP ALL SOUNDS ON ALL CHANNELS
SFX(-2)  --  RELEASE LOOPING ON ALL CHANNELS
```

MUSIC(N, [FADE_LEN], [CHANNEL_MASK])

Play music starting from pattern N (0..63)
N -1 to stop music

FADE_LEN is in ms (default: 0). So to fade pattern 0 in over 1 second:

```
MUSIC(0, 1000)
```

CHANNEL_MASK specifies which channels to reserve for music only. For example, to play only on channels 0..2:

MUSIC(0, NIL, 7) -- 1 | 2 | 4

Reserved channels can still be used to play sound effects on, but only when that channel index is explicitly requested by SFX().

6.6 Map

The PICO-8 map is a 128x32 grid of 8-bit values, or 128x64 when using the shared memory. When using the map editor, the meaning of each value is taken to be an index into the sprite sheet (0..255). However, it can instead be used as a general block of data.

MGET(X, Y)

MSET(X, Y, VAL)

Get or set map value (VAL) at X,Y

When X and Y are out of bounds, MGET returns 0, or a custom return value that can be specified with:

```
POKE(0x5f36, 0x10)
POKE(0x5f5a, NEWVAL)
```

MAP(TILE_X, TILE_Y, [SX, SY], [TILE_W, TILE_H], [LAYERS])

Draw section of map (starting from TILE_X, TILE_Y) at screen position SX, SY (pixels).

To draw a 4x2 blocks of tiles starting from 0,0 in the map, to the screen at 20,20:

```
MAP(0, 0, 20, 20, 4, 2)
```

TILE_W and TILE_H default to the entire map (including shared space when applicable).

MAP() is often used in conjunction with CAMERA(). To draw the map so that a player object (at PL.X in PL.Y in pixels) is centered:

```
CAMERA(PL.X - 64, PL.Y - 64)
MAP()
```

LAYERS is a bitfield. When given, only sprites with matching sprite flags are drawn. For example, when LAYERS is 0x5, only sprites with flag 0 and 2 are drawn.

Sprite 0 is taken to mean "empty" and is not drawn. To disable this behaviour, use: POKE(0x5F36, 0x8)

TLINE(X0, Y0, X1, Y1, MX, MY, [MDX, MDY], [LAYERS])

Draw a textured line from (X0,Y0) to (X1,Y1), sampling colour values from the map. When LAYERS is specified, only sprites with matching flags are drawn (similar to MAP())

MX, MY are map coordinates to sample from, given in tiles. Colour values are sampled from the 8x8 sprite present at each map tile. For example:

2.0, 1.0 means the top left corner of the sprite at position 2,1 on the map
2.5, 1.5 means pixel (4,4) of the same sprite

MDX, MDY are deltas added to mx, my after each pixel is drawn. (Defaults to 0.125, 0)

The map coordinates (MX, MY) are masked by values calculated by subtracting 0x0.0001 from the values at address 0x5F38 and 0x5F39. In simpler terms, this means you can loop a section of the map by poking the width and height you want to loop within, as long as they are powers of 2 (2,4,8,16..)

For example, to loop every 8 tiles horizontally, and every 4 tiles vertically:

```
POKE(0x5F38, 8)
POKE(0x5F39, 4)
TLINE(...)
```

The default values (0,0) gives a masks of 0xff.ffff, which means that the samples will loop every 256 tiles.

An offset to sample from (also in tiles) can also be specified at addresses 0x5f3a, 0x5f3b:

```
POKE(0x5F3A, OFFSET_X)
POKE(0x5F3B, OFFSET_Y)
```

Sprite 0 is taken to mean "empty" and not drawn. To disable this behaviour, use: POKE(0x5F36, 0x8)

Setting TLINE Precision

By default, tline coordinates (mx,my,mdx,mdy) are expressed in tiles. This means that 1 pixel is 0.125, and only 13 bits are used for the fractional part. If more precision is needed, the coordinate space can be adjusted to allow more bits for the fractional part. This can be useful for things like textured walls, where the accumulated error from mdx,mdy rounding maybe become visible when viewed up close.

The number of bits used for the fractional part of each pixel is stored in a special register that can be adjusted by calling TLINE once with a single argument:

TLINE(16) -- MX,MY,MDX,MDY expressed in pixels

6.7 Memory

PICO-8 has 3 types of memory:

1. Base RAM (64k): see layout below. Access with PEEK() POKE() MEMCPY() MEMSET()
2. Cart ROM (32k): same layout as base ram until 0x4300
3. Lua RAM (2MB): compiled program + variables

ⓘ

Technical note: While using the editor, the data being modified is in cart rom, but api functions such as **SPR**() and **SFX**() only operate on base ram. PICO-8 automatically copies cart rom to base ram (i.e. calls **RELOAD**()) in 3 cases:
1. When a cartridge is loaded
2. When a cartridge is run
3. When exiting any of the editor modes // can turn off with: poke(0x5f37,1)

□ Base RAM Memory Layout

```
0X0 □ □GFX
0X1000 GFX2/MAP2 (SHARED)
0X2000 MAP
0X3000 GFX FLAGS
0X3100 SONG
0X3200 SFX
0X4300 USER DATA
0X5600 CUSTOM FONT (IF ONE IS DEFINED)
0X5E00 PERSISTENT CART DATA (256 BYTES)
0X5F00 DRAW STATE
0X5F40 HARDWARE STATE
0X5F80 GPIO PINS (128 BYTES)
0X6000 SCREEN (8K)
0x8000 USER DATA
```

User data has no particular meaning and can be used for anything via **MEMCPY**(), **PEEK**() & **POKE**(). Persistent cart data is mapped to 0x5e00..0x5eff but only stored if **CARTDATA**() has been called. Colour format (gfx/screen) is 2 pixels per byte: low bits encode the left pixel of each pair. Map format is one byte per tile, where each byte normally encodes a sprite index.

□ Remapping Graphics and Map Data

The GFX, MAP and SCREEN memory areas can be reassigned by setting values at the following addresses:

```
0X5F54 GFX: □ □can be 0x00 (default) or 0x60 (use the screen memory as the
spritesheet)
0X5F55 SCREEN: can be 0x60 (default) or 0x00 (use the spritesheet as screen memory)
0X5F56 MAP: □ □can be 0x20 (default) or 0x10..0x2f, or 0x80 and above.
0X5F57 MAP SIZE: map width. 0 means 256. Defaults to 128.
```

Addresses can be expressed in 256 byte increments. So 0x20 means 0x2000, 0x21 means 0x2100 etc. Map addresses 0x30..0x3f are taken to mean 0x10..0x1f (shared memory area). Map data can only be contained inside the memory regions 0x1000..0x2fff, 0x8000..0xffff, and the map height is determined to be the largest possible size that fits in the given region.

GFX and SCREEN addresses can additionally be mapped to upper memory locations 0x80, 0xA0, 0xC0, 0xE0, with the constraint that MAP can not overlap with that address (in this case, the conflicting GFX and/or SCREEN mappings are kicked back to their default mapping).

ⓘ GFX and SCREEN memory mapping happens at a low level which also affects memory access functions (peek, poke, memcpy). The 8k memory blocks starting at 0x0 and 0x6000 can be thought of as pointers to a separate video ram, and setting the values at 0X5F54 and 0X5F56 alters those pointers.

PEEK(ADDR, [N])

Read a byte from an address in base ram. If N is specified, PEEK() returns that number of results (max: 8192). For example, to read the first 2 bytes of video memory:

```
A, B = PEEK(0x6000, 2)
```

POKE(ADDR, VAL1, VAL2, ...)

Write one or more bytes to an address in base ram. If more than one parameter is provided, they are written sequentially (max: 8192).

PEEK2(ADDR)

POKE2(ADDR, VAL)

PEEK4(ADDR)

POKE4(ADDR, VAL)

16-bit and 32-bit versions of PEEK and POKE. Read and write one number (VAL) in little-endian format:

- 16 bit: 0xffff.0000
- 32 bit: 0xffff.ffff

ADDR does not need to be aligned to 2 or 4-byte boundaries.

Alternatively, the following operators can be used to peek (but not poke), and are slightly faster:

```
@ADDR -- PEEK(ADDR)
%ADDR -- PEEK2(ADDR)
$ADDR -- PEEK4(ADDR)
```

MEMCPY(DEST_ADDR, SOURCE_ADDR, LEN)

Copy LEN bytes of base ram from source to dest. Sections can be overlapping

RELOAD(DEST_ADDR, SOURCE_ADDR LEN, [FILENAME])

Same as MEMCPY, but copies from cart rom.

The code section (>= 0x4300) is protected and can not be read.

If filename specified, load data from a separate cartridge. In this case, the cartridge must be local (BBS carts can not be read in this way).

CSTORE(DEST_ADDR, SOURCE_ADDR, LEN, [FILENAME])

Same as memcpy, but copies from base ram to cart rom.

CSTORE() is equivalent to CSTORE(0, 0, 0x4300)

The code section (>= 0x4300) is protected and can not be written to.

If FILENAME is specified, the data is written directly to that cartridge on disk. Up to 64 cartridges can be written in one session. See **Cartridge Data** for more information.

MEMSET(DEST_ADDR, VAL, LEN)

Write the 8-bit value VAL into memory starting at DEST_ADDR, for LEN bytes.

For example, to fill half of video memory with 0xC8:

```
> MEMSET(0x6000, 0xC8, 0x1000)
```

6.8 Math

MAX(X, Y)

MIN(X, Y)

MID(X, Y, Z)

Returns the maximum, minimum, or middle value of parameters

```
> ?MID(7,5,10) -- 7
```

FLR(X)

```
> ?FLR ( 4.1) --> 4
> ?FLR (-2.3) --> -3
```

CEIL(X)

Returns the closest integer that is equal to or below x

```
> ?CEIL( 4.1) --> 5
> ?CEIL(-2.3) --> -2
```

COS(X)

SIN(X)

Returns the cosine or sine of x, where 1.0 means a full turn. For example, to animate a dial that turns once every second:

```
FUNCTION _DRAW()
 CLS()
 CIRC(64, 64, 20, 7)
 X = 64 + COS(T()) * 20
 Y = 64 + SIN(T()) * 20
 LINE(64, 64, X, Y)
END
```

PICO-8's SIN() returns an inverted result to suit screenspace (where Y means "DOWN", as opposed to mathematical diagrams where Y typically means "UP").

```
> SIN(0.25) -- RETURNS -1
```

To get conventional radian-based trig functions without the y inversion, paste the following snippet near the start of your program:

```
P8COS = COS FUNCTION COS(ANGLE) RETURN P8COS(ANGLE/(3.1415*2)) END
P8SIN = SIN FUNCTION SIN(ANGLE) RETURN -P8SIN(ANGLE/(3.1415*2)) END
```

ATAN2(DX, DY)

Converts DX, DY into an angle from 0..1

As with cos/sin, angle is taken to run anticlockwise in screenspace. For example:

```
> ?ATAN(0, -1) -- RETURNS 0.25
```

ATAN2 can be used to find the direction between two points:

```
X=20 Y=30
FUNCTION _UPDATE()
 IF (BTN(0)) X-=2
 IF (BTN(1)) X+=2
 IF (BTN(2)) Y-=2
 IF (BTN(3)) Y+=2
END

FUNCTION _DRAW()
 CLS()
 CIRCFILL(X,Y,2,14)
 CIRCFILL(64,64,2,7)

 A=ATAN2(X-64, Y-64)
 PRINT("ANGLE: "..A)
 LINE(64,64,
  64+COS(A)*10,
  64+SIN(A)*10,7)
END
```

SQRT(X)

Return the square root of x

ABS(X)

Returns the absolute (positive) value of x

RND(X)

Returns a random number n, where 0 <= n < x

If you want an integer, use flr(rnd(x)). If x is an array-style table, return a random element between table[1] and table[#table].

SRAND(X)

Sets the random number seed. The seed is automatically randomized on cart startup.

```
FUNCTION _DRAW()
 CLS()
 SRAND(33)
 FOR I=1,100 DO
  PSET(RND(128),RND(128),7)
 END
END
```

Bitwise Operations

Bitwise operations are similar to logical expressions, except that they work at the bit level.

Say you have two numbers (written here in binary using the "0b" prefix):

```
X = 0b1010
Y = 0b0110
```

A bitwise AND will give you bits set when the corresponding bits in X /and/ Y are both set

```
> PRINT(BAND(X,Y)) -- RESULT:0B0010 (2 IN DECIMAL)
```

There are 9 bitwise functions available in PICO-8:

```
BAND(X, Y) -- BOTH BITS ARE SET
BOR(X, Y) -- EITHER BIT IS SET
BXOR(X, Y) -- EITHER BIT IS SET, BUT NOT BOTH OF THEM
BNOT(X) -- EACH BIT IS NOT SET
SHL(X, N) -- SHIFT LEFT N BITS (ZEROS COME IN FROM THE RIGHT)
SHR(X, N) -- ARITHMETIC RIGHT SHIFT (THE LEFT-MOST BIT STATE IS DUPLICATED)
LSHR(X, N) -- LOGICAL RIGHT SHIFT (ZEROS COMES IN FROM THE LEFT)
ROTL(X, N) -- ROTATE ALL BITS IN X LEFT BY N PLACES
ROTR(X, N) -- ROTATE ALL BITS IN X RIGHT BY N PLACES
```

Operator versions are also available: & | ^^ ~ << >> >>> <<> >><

For example: PRINT(67 & 63) -- result:3 equivalent to BAND(67,63)

Operators are slightly faster than their corresponding functions. They behave exactly the same, except that if any operands are not numbers the result is a runtime error (the function versions instead default to a value of 0).

Integer Division

Integer division can be performed with a \

```
> PRINT(9\2) -- RESULT:4 EQUIVALENT TO FLR(9/2)
```

6.9 Custom Menu Items

MENUITEM(INDEX, [LABEL], [CALLBACK])

Add or update an item to the pause menu.

INDEX should be 1..5 and determines the order each menu item is displayed.

LABEL should be a string up to 16 characters long

CALLBACK is a function called when the item is selected by the user. If the callback returns true, the pause menu remains open.

When no label or function is supplied, the menu item is removed.

```
MENUITEM(1, "RESTART PUZZLE",
 FUNCTION() RESET_PUZZLE() SFX(10) END
)
```

The callback takes a single argument that is a bitfield of L,R,X button presses.

```
MENUITEM(1, "FOO",
 FUNCTION(B) IF (B&1 > 0) THEN PRINTH("LEFT WAS PRESSED") END END
)
```

To filter button presses that are able to trigger the callback, a mask can be supplied in bits 0xff00 of INDEX. For example, to disable L, R for a particular menu item, set bits 0x300 in the index:

```
MENUITEM(2 | 0x300, "RESET PROGRESS",
  FUNCTION() DSET(0,0) END
)
```

Menu items can be updated, added or removed from within callbacks:

```
MENUITEM(3, "SCREENSHAKE: OFF",
  FUNCTION()
    SCREENSHAKE = NOT SCREENSHAKE
    MENUITEM(NIL, "SCREENSHAKE: "..(SCREENSHAKE AND "ON" OR "OFF"))
    RETURN TRUE -- DON'T CLOSE
  END
)
```

6.10 Strings and Type Conversion

Strings in Lua are written either in single or double quotes or with matching [[ ]] brackets:

```
S = "THE QUICK"
S = 'BROWN FOX';
S = [[
  JUMPS OVER
  MULTIPLE LINES
]]
```

The length of a string (number of characters) can be retrieved using the # operator:

```
>PRINT(#S)
```

Strings can be joined using the .. operator. Joining numbers converts them to strings.

```
>PRINT("THREE "..4) --> "THREE 4"
```

When used as part of an arithmetic expression, string values are converted to numbers:

```
>PRINT(2+"3") □ --> 5
```

TOSTR(VAL, [FORMAT_FLAGS])

Convert VAL to a string.

FORMAT_FLAGS is a bitfield:

□ 0x1: Write the raw hexadecimal value of numbers, functions or tables.
□ 0x2: Write VAL as a signed 32-bit integer by shifting it left by 16 bits.

TOSTR(NIL) returns "[nil]"

TOSTR() returns ""

```
TOSTR(17) □ □ □ -- "17"
TOSTR(17,0x1) □ -- "0x0011.0000"
TOSTR(17,0x3) □ -- "0x00110000"
TOSTR(17,0x2) □ -- "1114112"
```

TONUM(VAL, [FORMAT_FLAGS])

Converts VAL to a number.

```
TONUM("17.5") □-- 17.5
TONUM(17.5) □ □-- 17.5
TONUM("HOGE") □-- NO RETURN VALUE
```

FORMAT_FLAGS is a bitfield:

□ 0x1: Read the string as written in (unsigned, integer) hexadecimal without the "0x" prefix
□ □ □ □Non-hexadecimal characters are taken to be '0'.
□ 0x2: Read the string as a signed 32-bit integer, and shift right 16 bits.
□ 0x4: When VAL can not be converted to a number, return 0

```
TONUM("FF", □ □ □ 0x1) □-- 255
TONUM("1114112", □0x2) □-- 17
TONUM("1234abcd", 0x3) □-- 0x1234.abcd
```

CHR(VAL0, VAL1, ...)

Convert one or more ordinal character codes to a string.

When

```
CHR(64) □ □ □ □ □ □ □ □ □ □-- "@"
CHR(104,101,108,108,111) □ -- "hello"
```

ORD(STR, [INDEX], [NUM_RESULTS])

Convert one or more characters from string STR to their ordinal (0..255) character codes.

Use the INDEX parameter to specify which character in the string to use. When INDEX is out of range or str is not a string, ORD returns nil.

When NUM_RESULTS is given, ORD returns multiple values starting from INDEX.

```
ORD("@") □ □ □ □ -- 64
ORD("123",2) □ □ -- 50 (THE SECOND CHARACTER: "2")
ORD("123",2,3) □ -- 50,51,52
```

SUB(STR, POS0, [POS1])

Grab a substring from string str, from pos0 up to and including pos1. When POS1 is not specified, the remainder of the string is returned. When POS1 is specified, but not a number, a single character at POS0 is returned.

```
S = "THE QUICK BROWN FOX"
PRINT(SUB(S,5,9)) □ --> "QUICK"
PRINT(SUB(S,5)) □ □ --> "QUICK BROWN FOX"
PRINT(SUB(S,5,_)) □ --> "Q"
```

SPLIT(STR, [SEPARATOR], [CONVERT_NUMBERS])

Split a string into a table of elements delimited by the given separator (defaults to ","). When separator is a number n, the string is split into n-character groups. When convert_numbers is true, numerical tokens are stored as numbers (defaults to true). Empty elements are stored as empty strings.

```
SPLIT("1,2,3") □ □ □ □ □ □ □ -- {1,2,3}
SPLIT("ONE:TWO:3",":",FALSE) -- {"ONE","TWO","3"}
SPLIT("1,,2,") □ □ □ □ □ □ □ -- {1,"",2,""}
```

TYPE(VAL)

Returns the type of val as a string.

```
> PRINT(TYPE(3))
NUMBER
> PRINT(TYPE("3"))
STRING
```

6.11 Cartridge Data

Using **CARTDATA**(), **DSET**(), AND **DGET**(), 64 numbers (256 bytes) of persistent data can be stored on the user's PICO-8 that persists after the cart is unloaded or PICO-8 is shutdown. This can be used as a lightweight way to store things like high scores or to save player progress. It can also be used to share data across cartridges / cartridge versions.

If more than 256 bytes is needed, it is also possible to write directly to the cartridge using **CSTORE**(). The disadvantage is that the data is tied to that particular version of the cartridge. e.g. if a game is updated, players will lose their savegames. Also, some space in the data sections of the cartridge need to be left available to use as storage.

Another alternative is to write directly to a second cartridge by specifying a fourth parameter to **CSTORE**(). This requires a cart swap (which in reality only means the user needs to watch a spinny cart animation for 1 second).

```
CSTORE(0,0,0X2000, "SPRITE SHEET.P8")
-- LATER, RESTORE THE SAVED DATA:
RELOAD(0,0,0X2000, "SPRITE SHEET.P8")
```

CARTDATA(ID)

Opens a permanent data storage slot indexed by ID that can be used to store and retrieve up to 256 bytes (64 numbers) worth of data using **DSET**() and **DGET**().

```
CARTDATA("ZEP_DARK_FOREST")
DSET(0, SCORE)
```

ID is a string up to 64 characters long, and should be unusual enough that other cartridges do not accidentally use the same id. Legal characters are a..z, 0..9 and underscore (_)

Returns true if data was loaded, otherwise false.

CARTDATA can be called once per cartridge execution, and so only a single data slot can be used.

Once a cartdata ID has been set, the area of memory 0X5E00..0X5EFF is mapped to permanent storage, and can either be accessed directly or via **DGET**()/@DSET().

There is no need to flush written data -- it is automatically saved to permanent storage even if modified by directly **POKE**()'ing 0X5E00..0X5EFF.

DGET(INDEX)

Get the number stored at INDEX (0..63)

Use this only after you have called **CARTDATA**()

DSET(INDEX, VALUE)

Set the number stored at index (0..63)

Use this only after you have called **CARTDATA**()

6.12 GPIO

GPIO stands for "General Purpose Input Output", and allows machines to communicate with each other. PICO-8 maps bytes in the range 0x5f80..0x5fff to gpio pins that can be

POKE()ed (to output a value -- e.g. to make an LED light up) or @PEEK()ed (e.g. to read

the state of a switch).

GPIO means different things for different host platforms:

CHIP: □ □ □ □ 0x5f80..0x5f87 mapped to xio-p0..xio-p7
Pocket CHIP: □0x5f82..0x5f87 mapped to GPIO1..GPIO6
□ // xio-p0 & p1 are exposed inside the prototyping area inside the case.
Raspberry Pi: 0x5f80..0x5f9f mapped to wiringPi pins 0..31
□ // see http://wiringpi.com/pins/ for mappings on different models.
□ // also: watch out for BCM vs. WiringPi GPIO indexing!

CHIP and Raspberry Pi values are all digital: 0 (LOW) and 255 (HIGH)

A program to blink any LEDs attached on and off:

```
T = 0
FUNCTION _DRAW()
□CLS(5)
□FOR I=0,7 DO
□ VAL = 0
□ IF (T % 2 < 1) VAL = 255
□ POKE(0X5F80 + I, VAL)
□ CIRCFILL(20+I*12,64,4,VAL/11)
□END
□T += 0.1
END
```

□ Serial

For more precise timing, the **SERIAL**() command can be used. GPIO writes are buffered and dispatched at the end of each frame, allowing clock cycling at higher and/or more regular speeds than is possible by manually bit-banging using **POKE**() calls.

SERIAL(CHANNEL, ADDRESS, LENGTH)

CHANNEL:
□ 0x000..0x0fe □ □corresponds to gpio pin numbers; send 0x00 for LOW or 0xFF for HIGH
□ 0x0ff □ □ □ □ □ delay; length is taken to mean "duration" in microseconds (excl. overhead)
□ 0x400..0x401 □ □ws281x LED string (experimental)

ADDRESS: The PICO-8 memory location to read from / write to.

LENGTH: Number of bytes to send. 1/8ths are allowed to send partial bit strings.

For example, to send a byte one bit at a time to a typical APA102 LED string:

```
VAL = 42 □ □ □ □ □-- VALUE TO SEND
DAT = 16 CLK = 15 -- DATA AND CLOCK PINS DEPEND ON DEVICE
POKE(0X4300,0) □ □-- DATA TO SEND (SINGLE BYTES: 0 OR 0XFF)
POKE(0X4301,0XFF)
FOR B=0,7 DO
□ -- SEND THE BIT (HIGH FIRST)
□ SERIAL(DAT, BAND(VAL, SHL(1,7-B))>0 AND 0X4301 OR 0X4300, 1)
□ -- CYCLE THE CLOCK
□ SERIAL(CLK, 0X4301)
□ SERIAL(0XFF, 5) -- DELAY 5
□ SERIAL(CLK, 0X4300)
□ SERIAL(0XFF, 5) -- DELAY 5
END
```

Additional channels are available for bytestreams to and from the host operating system. These are intended to be most useful for UNIX-like environments while developing toolchains, and are not available while running a BBS or exported cart [1]. Maximum transfer rate in all cases is 64k/sec (blocks cpu).

```
0x800 □dropped file □ // □stat(120) returns TRUE when data is available
0x802 □dropped image □// □stat(121) returns TRUE when data is available
0x804 □stdin
0x805 □stdout
0x806 □file specified with: pico8 -i filename
0x807 □file specified with: pico8 -o filename
```

Image files dropped into PICO-8 show up on channel 0x802 as a bytestream with a special format: The first 4 bytes are the image's width and height (2 bytes each little-endian, like PEEK2), followed by the image in reading order, one byte per pixel, colour-fitted to the display palette at the time the file was dropped.

[1] Channels 0x800 and 0x802 are available from exported binaries, but with a maximum file size of 256k, or 128x128 for images.

□ HTML

Cartridges exported as HTML / .js use a global array of integers (pico8_gpio) to represent gpio pins. The shell HTML should define the array:

```
var pico8_gpio = Array(128);
```

6.13 Mouse and Keyboard Input

```
// EXPERIMENTAL -- but mostly working on all platforms
```

Mouse and keyboard input can be achieved by enabling devkit input mode:

```
POKE(0x5F2D, flags) -- where flags are:

0x1 Enable
0x2 Mouse buttons trigger btn(4)..btn(6)
0x4 Pointer lock (use stat 38..39 to read movements)
```

Note that not every PICO-8 will have a keyboard or mouse attached to it, so when posting carts to the Lexaloffle BBS, it is encouraged to make keyboard and/or mouse control optional and off by default, if possible. When devkit input mode is enabled, a message is displayed to BBS users warning them that the program may be expecting input beyond the standard 6-button controllers.

The state of the mouse and keyboard can be found in stat(x):

```
STAT(30) -- (Boolean) True when a keypress is available
STAT(31) -- (String) character returned by keyboard
STAT(32) -- Mouse X
STAT(33) -- Mouse Y
STAT(34) -- Mouse buttons (bitfield)
STAT(36) -- Mouse wheel event
STAT(38) -- Relative x movement (in host desktop pixels) -- requires flag 0x4
STAT(39) -- Relative y movement (in host desktop pixels) -- requires flag 0x4
```

6.14 Additional Lua Features

PICO-8 also exposes 2 features of Lua for advanced users: Metatables and Coroutines.

For more information, please refer to the Lua 5.2 manual.

□ Metatables

Metatables can be used to define the behaviour of objects under particular operations. For example, to use tables to represent 2D vectors that can be added together, the '+' operator is redefined by defining an "__add" function for the metatable:

```
VEC2D={
 __ADD=FUNCTION(A,B)
  RETURN {X=(A.X+B.X), Y=(A.Y+B.Y)}
 END
}

V1={X=2,Y=9} SETMETATABLE(V1, VEC2D)
V2={X=1,Y=5} SETMETATABLE(V2, VEC2D)
V3 = V1+V2
PRINT(V3.X..","..V3.Y) -- 3,14
```

SETMETATABLE(TBL, M)

Set table TBL metatable to M

GETMETATABLE(TBL)

return the current metatable for table t, or nil if none is set

RAWSET(TBL, KEY, VALUE)

RAWGET(TBL, KEY)

RAWEQUAL(TBL1,TBL2

RAWLEN(TBL)

Raw access to the table, as if no metamethods were defined.

Function Arguments

The list of function arguments can be specifed with ...

```
FUNCTION PREPRINT(PRE, S, ...)
 LOCAL S2 = PRE..TOSTR(S)
 PRINT(S2, ...) -- PASS THE REMAINING ARGUMENTS ON TO PRINT()
END
```

To accept a variable number of arguments, use them to define a table and/or use Lua's select() function. select(index, ...) returns all of the arguments after index.

```
FUNCTION FOO(...)
 LOCAL ARGS={...} -- BECOMES A TABLE OF ARGUMENTS
 FOREACH(ARGS, PRINT)
 ?SELECT("#",...)  --ALTERNATIVE WAY TO COUNT THE NUMBER OF ARGUMENTS
 FOO2(SELECT(3,...)) -- PASS ARGUMENTS FROM 3 ONWARDS TO FOO2()
END
```

Coroutines

Coroutines offer a way to run different parts of a program in a somewhat concurrent way, similar to threads. A function can be called as a coroutine, suspended with

YIELD() any number of times, and then resumed again at the same points.

```
FUNCTION HEY()
 PRINT("DOING SOMETHING")
 YIELD()
 PRINT("DOING THE NEXT THING")
 YIELD()
 PRINT("FINISHED")
END

C = COCREATE(HEY)
FOR I=1,3 DO CORESUME(C) END
```

COCREATE(F)

Create a coroutine for function f.

CORESUME(C, [P0, P1 ..])

Run or continue the coroutine c. Parameters p0, p1.. are passed to the coroutine's function.

Returns true if the coroutine completes without any errors Returns false, error_message if there is an error.

** Runtime errors that occur inside coroutines do not cause the program to stop running. It is a good idea to wrap CORESUME() inside an **ASSERT**(). If the assert fails, it will print the error message generated by coresume.

ASSERT(CORESUME(C))

COSTATUS(C)

Return the status of coroutine C as a string:
- "running"
- "suspended"
- "dead"

YIELD

Suspend execution and return to the caller.

❼ Appendix

7.1 Appendix A: P8SCII Control Codes

When printed with **PRINT**(), some characters have a special meaning that can be used to alter things like the cursor position and text rendering style. Control characters in PICO-8 are CHR(0)..CHR(15) and can be written as an escaped sequence ("\n" for newline etc.)

Some of the control codes below take parameters which are written using a scheme that is a superset of hexadecimal format. That is, '0'..'f' also mean 0..15. But characters after 'f' are also accepted: 'g' means 16 and so on. Such parameters are written below as P0, P1.

For example, to print with a blue background ("\#c") and dark gray foreground ("\f5"):

```
PRINT("\#C\F5 BLUE ")
```

The only side-effects on the draw state are changes in cursor position and foreground color; all other attributes are reset each time **PRINT**() is called.

□ Control Codes

```
0 "\0" □ terminate printing
1 "\*" □ repeat next character P0 times. ?"\*3a" --> aaa
2 "\#" □ draw solid background with colour P0
3 "\-" □ shift cursor horizontally by P0-16 pixels
4 "\|" □ shift cursor vertically by P0-16 pixels
5 "\+" □ shift cursor by P0-16, P1-16 pixels
6 "\^" □ special command (see below)
7 "\a" □ audio (see below)
8 "\b" □ backspace
9 "\t" □ tab
a "\n" □ newline
b "\v" □ decorate previous character (see below)
c "\f" □ set foreground colour
d "\r" □ carriage return
e "\014" switch to font defined at 0x5600
f "\015" switch to default font
```

□ Special Commands

These commands all start with "\^" and take up to 2 parameters (P0, P1) For example, to clear screen to dark blue: print("\^c1")

```
1..9 skip 1,2,4,8,16,32..256 frames
c cls to colour P0, set cursor to 0,0
d set delay to P0 frames for every character printed
g set cursor position to home
h set home to cursor position
j jump to absolute P0*4, P1*4 (in screen pixels)
r set rhs character wrap boundary to P0*4
s set tab stop width to P0 pixels (used by "\t")
x set character width □(default: 4)
y set character height (default: 6)
```

□ Rendering mode options

```
// prefix these with "-" to disable: e.g. ?"\^i on \^-i off "

w wide mode: scales by 2x1
t tall mode: scales by 1x2
= stripey mode: when wide or tall, draw only even pixels
p pinball mode: equivalent to setting wide, tall and stripey
i invert
b border: toggle 1px padding on left and top // on by default
# solid background □// off by default, but enabled automatically by \#
```

□ Raw memory writes

The following two commands take 4-character hex parameters:

@addrnnnn[binstr] poke nnnn bytes to address addr
!addr[binstr] □ □ poke all remaining characters to address addr

For example, to write 4 bytes to video memory halfway down the screen:

```
>?"\^@70000004xxxxhello"
```

□ One-off characters

Character data can be specified and printed in-line using \^. followed by 8 bytes of raw binary data, or \^: followed by 8 2-digit hexadecimal values. The data format is the same as custom fonts; each byte specifies a row of 1-bit pixel values, with the low bit on the left.

\^.[8 chars of raw binary data]
\^:[16 chars of hexadecimal]

To print a cat:

```
> ?"\^:447cb67c3e7f0106"
```

□ Audio

? ?"\A" -- SINGLE BEEP ?"\A12" -- PLAY EXISTING DATA AT SFX 12

If an sfx index is not specified, a non-active sfx between 60..63 is selected automatically. To fill the SFX with data before playback, the following commands can then be appended.

1. (optional) SFX attributes must appear once at the start as they apply to the whole sound:

s P0 □ □ set the sfx speed
l P0 P1 □set the sfx loop start and end points

2. Note data:

Note are written as a..g, optionally followed by a sharp # or flat -, and octave number.

```
PRINT "\ACE-G" -- MINOR TRIAD
```

Empty notes Can be written with a dot:

```
PRINT "\AC..E-..G" -- STACCATO MINOR TRIAD
```

Note attribute commands apply to following notes:

i P0 □ □set the instrument (default: 5)
v P0 □ □set the volume □ □ (default: 5)
x P0 □ □set the effect □ □ (default: 0)
< > increase or decrease volume by 1

For example, to play a fast (speed 4), staccato (effect 5) arpeggio starting at C1:

```
PRINT "\AS4X5C1EGC2EGC3EGC4"
```

□ Decoration Characters

The control character \v can be used to decorate the last printed character with another character at a given offset, without needing to otherwise manage the cursor position. After the decorating character is printed, the previous cursor position is restored.

The format is \v P0 char, where P0 is a number giving the desired offset, and char is any character to print at that offset (relative to the previous printed character).

The offset has x packed into the lowest 2 bits, and starts (-2,-8) in reading order. So 3 means (+1, -8), 4 means (-2, -7) and so on.

For example, to write "caf□□!", using a comma to draw the acute accent:

```
PRINT"\NCAFE\VB,!"
```

In this case P0 is 'b', which is read as the number 11. So the comma is drawn at:

x = (11%4)-2 = 1
y = (11\4)-8 = -6

□ Custom Font

A custom font can be defined at 0x5600, consisting of 8 bytes per character * 256 characters = 2048 bytes. Each character is an 8x8 bitfield (1 bit/pixel), where starting from the top, each row is a single byte starting with 0x1 on the left.

The first 128 bytes (characters 0~15 are never drawn) describe attributes of the font:

```
0x5600 character width in pixels (can be more than 8, but only 8 pixels are drawn)
0x5601 character width for character 128 and above
0x5602 character height in pixels
0x5603 draw offset x
0x5604 draw offset y
0x5605 flags: 0x1 apply_size_adjustments 0x2: apply tabs relative to cursor home
0x5606 tab width in pixels (used only when alt font is drawn)
0x5607 unused
```

The remaining 120 bytes are used to adjust the width and vertical offset of characters 16..255. Each nibble (low nibbles first) describes the adjustments for one characters:

```
bits 0x7: adjust character width by 0,1,2,3,-4,-3,-2,-1
bit 0x8: when set, draw the character one pixel higher (useful for latin accents)
```

Default Attributes

Although attributes are reset every time **PRINT**() is called, it is possible to set their default values by writing to memory addresses 0x5f58..0x5f5b.

```
0x5f58 // bitfield
 0x1 when set to 0x1, bits 1..7 are observed:
 0x2 padding
 0x4 wide
 0x8 tall
 0x10 solid background
 0x20 invert
 0x40 stripey (when wide or tall)
 0x80 use custom font

 // e.g. poke(0x5f58, 0x1 | 0x2 | 0x4 | 0x8 | 0x20 | 0x40) -- pinball everywhere

0x5f59 char_w  (low nibble), char_h  (high)
0x5f5a char_w2 (low nibble), tab_w  (high)
0x5f5b offset_x (low nibble), offset_y (high)

// any nibbles equal to 0 are ignored
// tab_w (global tab width) values are mapped to 4..60
```