MINI micro

v1.0

the neo-retro classic-modern home computer from an alternate universe

MiniScript at the Prompt

*You can type any MiniScript commands at the "**]**" command prompt.*

```
]print "Hello world!"
Hello world!
```

See the last page of this document for a quick rundown on the MiniScript language. Or go to http://miniscript.org for more help.

*Press **up arrow** to recall the last command. When more input is needed, the prompt will change to "...]". Press **Control-C** to break an infinite loop or reset the prompt.*

Basic Commands

clear	clear/reset display
help	get online help
pprint *value*	pretty-print a map or list

Disk and Files

There are usually two disks available, "/sys" and "/usr". /sys is the system disk; it contains demos, game assets, libraries, etc. It is a read-only disk; you cannot modify its contents. /usr is the user disk; you can use it however you like. This is where you will store your own MiniScript programs. Click the top disk slot to create a new disk, or mount a zip file or folder as /usr.

*Remember that the command prompt runs MiniScript, not some other shell. So **you must use quotation marks around file names and paths** in all commands.*

Global File Commands

pwd	print working directory
cd *path*	change working directory
dir	list files
mkdir *path*	create a new directory
delete *path*	delete a file from disk
view *path*	preview any file

File module

The global file module contains more methods for working with files and paths. Use these like file.curdir, etc:

.curdir	return working directory
.setdir *path*	same as cd
.makedir *path*	create a new directory
.children(path)	get files within directory
.name*(path)*	get file name from path
.parent*(path)*	get path to parent directory
.exists*(path)*	return whether file exists
.info*(path)*	get map of file details
.child*(base, subpath)* — combine path parts	
.delete *path*	delete a file
.move *from, to*	move/rename a file
.copy *from, to*	copy a file
.readLines(*path*)	return file contents as list
.writeLines *path, list* — store list as text file	
.loadImage(*path*) — load an image file	
.saveImage *path, img, [quality]*	
.loadSound(*path*) — load a sound file	
.export *path*	export file to host OS
.import *path*	import file from host OS
.open*(path, mode)* — return a file handle	

File Handle

A file handle object is returned from file.open, and is used for more detailed input and output with a particular file.

.isOpen	is the file still open?
.position	get/set read/write position
.atEnd	is position at end of file?
.write *s*	write string to file
.writeLine *s*	write string followed by EOL
.read *[bytes]*	return file data as string
.readLine	return next line of file
.close	close the file when done

Handling Programs

Mini Micro has one "current program" in memory at a time. The commands below let you load, save, edit, run, or clear this program.

load *filename*	load a program
source	show source code listing
run	run current program
edit	edit current program
save *[path]*	save program to disk
reset	clear program from memory

The code editor (invoked with edit) has a lot of nice features, both in the toolbar and via keyboard shortcuts. Try it!

["Getting Started" Example]()

```
cd "/sys/demo"
dir
load "ticTacToe"
run

clear
```

Key & Mouse Input

input(prompt)	return a line of user input
key.available	is there a key in the buffer?
key.get	return next key pressed
key.clear	clear the key buffer
key.pressed(k)	is key k currently pressed?
key.keyNames	all names for key.pressed
key.axis(h)	value of analog axis h
mouse.x	current mouse X position
mouse.y	current mouse Y position
mouse.button(which=0) — return whether the given mouse button is pressed	
mouse.visible	show the mouse cursor?

Key names for key.pressed are shown in the table below. Axis names are "Horizontal", "Vertical", and "Joy1Axis1" through "Joy8Axis29". Note that "joystick" refers to any game input device (gamepad, flight stick, etc.).

Key names for key.pressed	
normal keys	`"a", "b", "c", …`
number keys	`"1", "2", "3", …`
arrow keys	`"up", "down", "left", "right"`
keypad keys	`"[1]", "[2]", "[3]", … "[+]", "[-]", "[/]", "[*]"`
function keys	`"f1", "f2", "f3", …`
modifier keys	`"left shift", "right shift", "left ctrl", "right ctrl", "left alt", "right alt", "left cmd", "right cmd"`
special keys	`"backspace", "tab", "return", "escape", "space", "delete", "enter", "insert", "home", "end", "page up", "page down"`
mouse buttons	`"mouse 0", "mouse 1", …`
joystick buttons (any joystick)	`"joystick button 0", "joystick button 1", …`
buttons on a specific joystick	`"joystick 1 button 0", "joystick 2 button 0", …`

About this Document

*Property names shown in **orange** can be read or assigned new values, like any variable:*

```
text.row = 25
```

*Method names shown in **blue** can be called and may return results, but you don't assign new values to them:*

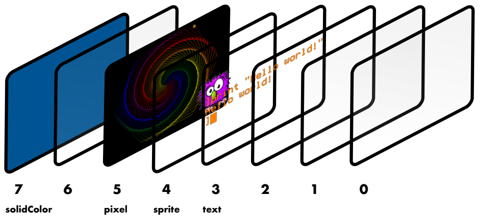
```
text.clear
print text.cell(0,0)
```

Displays

Mini Micro has an 8-layer display. Display 0 is closest to the user; display 7 in is the back. You can see through transparent displays to any higher-numbered display layers behind. Each display can be one of several modes:

0. displayMode.off	hidden/off
1. displayMode.solidColor	solid color
2. displayMode.text	text display
3. displayMode.pixel	pixel buffer
4. displayMode.tile	tile display
5. displayMode.sprite	sprite display

The default setup is shown in the diagram below. Change any display by assigning one of the above values to display(n).mode, where n is from 0 to 7. Then get a reference to display(n), and use the methods on the appropriate Display subclass.



Solid Color Display

Simply displays the same color across the whole screen. Translucent colors work too. Useful for fade in/out or as background.

.color	display color

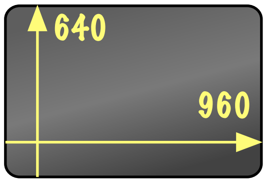
Text Display

A 68-by-26 character display. Every cell may have its own colors and inverse mode; the properties below mostly affect subsequent printing. Note that **text** *is a global reference to the "default" text display, i.e., the one used by* **print** *and* **input**.

.color	text color (for later **print**)
.backColor	background color
.column, .row	cursor column and row
.inverse	when true, swap colors
.delimiter	follows every **print**
.clear	clears the display
.cell(x,y)	get character at col x, row y
.setCell x, y, k	stuff k into col x, row y
.cellColor(x,y)	get text color in a given cell
.setCellColor x, y, c — set text color	
.cellBackColor(x,y) — get background color	
.setCellBackColor x, y, c — set bkgnd color	
.print s	print to this display

Note that the standard text.delimiter is char(13)*, which causes a line break. Use "" (empty string) for no delimiter.*

Pixel Display



A 960-by-640 display made of pixels. **gfx** *is a handy reference to the default pixel display.*

.color	default drawing color
.width, .height	get display size, in pixels
.clear *[clr, w, h]*	fill display with given color
.pixel(x,y)	get pixel color at x,y
.setPixel x, y, clr	set pixel color at x,y
.scrollX, .scrollY	offsets display in X and Y
.scale	scale factor or [x,y] factors

The drawing methods below all do what they say. Not shown here are two optional parameters: color and penSize.

.line x1, y1, x2, y2
.drawRect left, bottom, width, height
.fillRect left, bottom, width, height
.drawEllipse left, bottom, width, height
.fillEllipse left, bottom, width, height
.drawPoly points
.fillPoly points

The functions below work with the Image class:

.drawImage img, left, bottom, width, height,
srcLeft, srcBottom, srcWidth, srcHeight
.getImage(left, bottom, width, height)

The .print method draws text to a pixel display; this is slower than using a text display, but more versatile. Available fonts are "small", "normal", and "large".

.print str, x, y, color, font="normal"

Tile Display

A tile display shows a rectangular or hexagonal grid of small images called tiles. You can configure the size of and number of these tiles, their overlap, and an overall scroll position.

.clear *[toIndex]*	set all tiles to null or index
.extent	[cols, rows] map size
.tileSet	image tiles draw from
.tileSetTileSize	size of tiles in tileSet
.cellSize	size of tiles on screen
.overlap	cell overlap, in pixels
.oddRowOffset	set to 0.5 for hex rows
.oddColOffset	set to 0.5 for hex columns
.scrollX, .scrollY	shifts all tiles on screen
.cell(x,y)	get tile index for a cell
.setCell x, y, idx	set tile index for a cell
.cellTint(x,y)	get tint color of a cell
.setCellTint x, y, c	set tint color of a cell

Some tile display properties (extent, tileSetTileSize, cellSize, and overlap) can be given either a simple number which applies to both x and y, or an [x,y] list.

Sprite Display

Each sprite display shows 0 or more Sprites, which are little images that can be efficiently moved, rotated, and scaled. Sprites are layered in order, with .sprites[0] at the back.

.clear	removes all sprites
.sprites	list of sprites to draw
.scrollX, .scrollY	shifts all sprites on screen

Sprite Class

.image	image (see file.loadImage)
.x, .y	position of sprite on screen
.scale	scale factor or [x,y] factors
.rotation	angle in degrees
.tint	tint color (white for no tint)
.localBounds	bounds relative to this sprite
.worldBounds	returns bounds on screen
.contains(*pt*)	bounds-containment test
.overlaps(*other*)	bounds-touching test

Bounds Class

.x, .y	center of bounding box
.width, .height	bounding box size
.rotation	angle in degrees
.corners	returns box corners as list
.overlaps *b*	is this box touching box b?
.contains *x,y*	is point x,y within this box?

The .contains method (of both Bounds and Sprite) may also be given any map with "x" and "y" keys, or an [x, y] list.

Image Class

Represents a rectangular array of pixels; display with either Sprite.image, or PixelDisplay.drawImage. Methods:

.width, .height	image size, in pixels
.pixel(x,y)	get pixel color at x,y
.setPixel x, y, clr	set pixel color at x,y
.getImage(left, bottom, width, height)	

Create an image from scratch with:

Image.create(width, height, color)

Colors

Colors in Mini Micro are represented as strings in HTML format. The **color** *map contains the built-in colors shown below, as well as these methods:*

.rgb(r, g, b)	get color from red, green, and blue values (0-255)
.rgba(r, g, b, a)	same, but with alpha
.lerp(c1, c2, t)	interpolate between colors
.toList(c)	get color as [r, g, b, a] list
.fromList(lst)	convert [r, g, b, a] to color

aqua	🟦	navy	🟦
black	⬛	olive	🟩
blue	🟦	orange	🟧
brown	🟫	pink	🟥
clear		purple	🟪
fuchsia	🟪	red	🟥
gray	⬜	silver	⬜
green	🟩	teal	🟦
lime	🟩	white	
maroon	🟥	yellow	🟨

Sounds

*Mini Micro supports both digitized and synthesized sounds via the **Sound** class. Use the **file** module to load a sound from disk:*

file.loadSound	load a WAV file as a sound

To create a synthesized sound, make a new Sound object, then set the following properties:

.duration	sound length (sec)
.freq	frequency (Hz)
.envelope	volume over time (0-1)
.waveform	one cycle of sound wave
.fadeIn	length of fade-in (sec)
.fadeOut	length of fade-out (sec)
.loop	set to 1 to loop until stopped

*You can conveniently set duration, freq, envelope, and waveform with the **.init** method on the Sound class.*

Frequency

The .freq property determines how many times per second the waveform will be repeated. The "A" above middle C on a piano has a frequency of 440. A global method provides the frequency for any note:

noteFreq(n)	frequency for note n

Middle C is note 60, C# is 61, etc.

Instead of specifying a single frequency, you can provide a list of frequencies; Mini Micro will then interpolate (slide) between those frequencies over the length of the sound.

Envelope

The .envelope property controls the amplitude (volume) of the sound over its duration. You may specify a single number (the default is 1), or a list of numbers, in which case Mini Micro will interpolate the amplitude over the length of the sound. A common choice is [1, 0] which starts at full volume and then fades to silence by the end of the sound.

Waveform

The .waveform property determines the tonal quality of the sound. This should be a list of numbers between -1 and 1. Mini Micro will interpolate over this list for <u>each</u> repeat of the waveform — if freq is 440, the waveform will be repeated 440 times per second.

The Sound class has several built-in waveforms for your convenience:

.sineWave	sine wave (pure tone)
.triangleWave	triangles (almost sine)
.sawtoothWave	slightly "buzzier"
.squareWave	most buzzy/retro sound
.noiseWave	random static

Sound Mixing

You can combine two or more synthesized sounds together to create more complex sounds.

.mix(s2, *lvl=1*)	add in sound s2, at volume level lvl

Playing Sounds

Both digitized and synthesized sounds are played with the .play method:

.play *v, p, s*	play sound at volume v, with pan p and speed s

All parameters optional. Volume should be between 1 and 0; pan between -1 and 1 (full left/right); and speed is a multiplier that changes the playback speed and pitch (default is 1).

Other methods on Sound objects:

.stop	stop playing this sound
.isPlaying	is sound currently playing?

Silence all sounds at once with:

Sound.stopAll	stop all sounds

HTTP

The http module provides simple access to downloading resources or making REST calls on the interwebs.

.get(url, *headers*)	download
.delete url, *headers*	delete resource
.post url, data, *headers*	post data to a URL
.put url, data, *headers*	do an HTTP PUT

http.get can download images, sounds, text, or raw data. http.post data may be a string or a map.

Silly Sketch Example

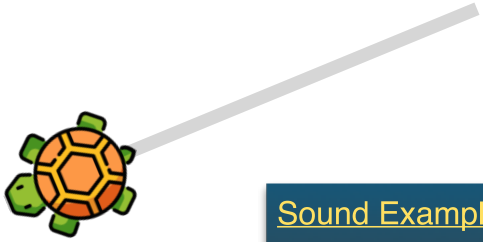
```
clear
text.row = 25
print "Draw with the mouse!"
print "Press Esc to exit."
snd = new Sound

while not key.pressed("escape")
   m = {}
   m.x = mouse.x
   m.y = mouse.y
   if mouse.button then
      gfx.line prev.x, prev.y,
       m.x, m.y, color.gray, 5
      snd.init 0.1, 400 + m.y
      snd.play 0.5
   end if
   prev = m
   yield
end while
```

Import Modules

There are a number of handy utilities found in /sys/lib, which you can load with the import command:

import *"name"*	find & load module by name

These modules can define new values and methods (accessed via a map with the same name of the module), and add new methods to built-in types. For more info, see: help "import"



Sound Example 1

```
pew = new Sound
pew.init 0.3, [8000,100], [1,0]
pew.play
```

Sound Example 2

```
hitSnd = new Sound
hitSnd.init 1, 100, [1,0], Sound.noiseWave
hitSnd.play
```

Music Example

```
// notes defined as: [note, duration]
notes = [[60, 0.1], [64, 0.1], [67, 0.1],
         [72, 0.2], [67, 0.1], [72, 0.4]]
snd = new Sound
for n in notes
   snd.init n[1], noteFreq(n[0])
   snd.play
   wait snd.duration
end for
```

(The MiniScript Quick Reference appears on the next page for your convenience.)

Welcome to MiniScript!

MiniScript is a high-level object-oriented language that is easy to read and write.

Clean Syntax

Put one statement per line, with no semicolons, except to join multiple statements on one line.

Code blocks are delimited by keywords (see below). Indentation doesn't matter (except for readability).

*Comments begin with **//**.*

*Don't use empty parentheses on function calls, or around conditions in **if** or **while** blocks.*

All variables are local by default. MiniScript is case-sensitive.

Control Flow

if, else if, else, end if

*Use **if** blocks to do different things depending on some condition. Include zero or more **else if** blocks and one optional **else** block.*

```
if 2+2 == 4 then
   print "math works!"
else if pi > 3 then
   print "pi is tasty"
else if "a" < "b" then
   print "I can sort"
else
   print "last chance"
end if
```

while, end while

*Use a **while** block to loop as long as a condition is true.*

```
s = "Spam"
while s.len < 50
   s = s + ", spam"
end while
print s + " and spam!"
```

for, end for

*A **for** loop can loop over any list, including ones easily created with the **range** function.*

```
for i in range(10, 1)
   print i + "..."
end for
print "Liftoff!"
```

break & continue

*The **break** statement jumps out of a **while** or **for** loop. The **continue** statement jumps to the top of the loop, skipping the rest of the current iteration.*

Data Types

Numbers

All numbers are stored in full-precision format. Numbers also represent true (1) and false (0). Operators:

+, -, *, /	standard math
%	mod (remainder)
^	power
and, or, not	logical operators
==, !=, >, >=, <, <=	comparison

Strings

Text is stored in strings of Unicode characters. Write strings by surrounding them with quotes. If you need to include a quotation mark in the string, type it twice.

```
print "OK, ""Bob""."
```

Operators:

+	string concatenation
-	string subtraction (chop)
*, /	replication, division
==, !=, >, >=, <, <=	comparison
[i]	get character i
[i:j]	get slice from i up to j

Lists

*Write a list in square brackets. Iterate over the list with **for**, or pull out individual items with a 0-based index in square brackets. A negative index counts from the end. Get a slice (subset) of a list with two indices, separated by a colon.*

```
x = [2, 4, 6, 8]
x[0]    // 2
x[-1]   // 8
x[1:3]  // [4, 6]
x[2]=5  // x now [2,4,5,8]
```

Operators:

+	list concatenation
*, /	replication, division
[i]	get/set element i
[i:j]	get slice from i up to j

Maps

A map is a set of values associated with unique keys. Create a map with curly braces; get or set a single value with square brackets. Keys and values may be any type.

```
m = {1:"one", 2:"two"}
m[1]    // "one"
m[2] = "dos"
```

Operators:

+	map concatenation
[k]	get/set value with key k
.ident	get/set value by identifier

Functions

*Create a function with **function()**, including parameters with optional default values. Assign the result to a variable. Invoke by using that variable. Use @ to reference a function without invoking.*

```
triple = function(n=1)
   return n*3
end function
print triple      // 3
print triple(5)   // 15
f = @triple
print f(5)    // also 15
```

Classes & Objects

*MiniScript uses prototype-based inheritance. A class or object is a map with a special **__isa** entry that points to the parent. This is set automatically when you use the **new** operator.*

```
Shape = {"sides":0}
Square = new Shape
Square.sides = 4
x = new Square
x.sides   // 4
```

*Functions invoked via dot syntax get a **self** variable that refers to the object they were invoked on.*

```
Shape.degrees = function()
   return 180*(self.sides-2)
end function
x.degrees    // 360
```

Intrinsic Functions

Numeric

```
abs(x)      acos(x)   asin(x)
atan(x)     ceil(x)   char(i)
cos(r)      floor(x)  log(x,b)
round(x,d)  rnd       rnd(seed)
pi          sign(x)   sin(r)
sqrt(x)     str(x)    tan(r)
```

String

```
.hasIndex(i)   .indexOf(s)
.len        .val      .code
.remove(s)  .lower    .upper
.replace(a,b)         .split(d)
```

List/Map

```
.hasIndex(i)   .indexOf(x)
.indexes    .values   .join(s)
.len        .sum      .sort
.shuffle    .remove(i)
.push(x)    .pop      .pull
range(from,to,step)
```

Other

```
print(s)    time      wait(sec)
locals      globals   yield
```