

# Inkspector 2.0.7

## User Manual

### Revision 1

#### Table of Contents

Introduction.....	9
Main Window.....	10
File Menu.....	10
Edit Menu.....	13
Display Menu.....	13
View Menu.....	13
Machine Menu.....	13
Tools Menu.....	14
Search Library.....	14
Keyboard Assist.....	15
Add Breakpoint.....	17
Poke Memory.....	20
Poke Manager.....	21
RZX Verifier.....	21
View BASIC Program.....	22
View User Variables.....	23
View Machine State.....	24
Spool Clipboard, Spool File.....	24
Assemble Clipboard Contents.....	25
Recording.....	25
Record Audio.....	25
Record AV.....	25
Record Animated GIF.....	25
Recording Indicator.....	25
Stop Script.....	26
Save Configuration.....	26
Options.....	26
Options Screen.....	26
Start-up and Shut Down.....	26
Start-up.....	26
Shut Down.....	27
ZX80, ZX81, Jupiter ACE.....	27
ZX Spectrum.....	29
Timings.....	29
Spectrum 16K, 48K.....	29
Spectrum 128.....	30
Spectrum +3.....	30
Kempston Joystick.....	30
ZX Printer.....	30
ZX Interface 1.....	31

Floppy Disks.....	31
Peripherals.....	32
Keyboard.....	34
COMCON Programmable Joystick Interface.....	34
Joysticks.....	35
Display.....	36
Audio.....	36
Snapshots.....	37
SZX Snapshots.....	37
Z80 Snapshots.....	37
Tapes.....	38
Recordings.....	39
Animated GIFs.....	39
RZX Recordings.....	39
AV Recording.....	40
ROM Management (including listing and symbol files).....	40
Assembler.....	44
Confirmation.....	44
Errors.....	44
Auxiliary Snapshot Files.....	45
Debugger.....	46
Symbols and Source Code.....	46
Disassembly.....	47
Breakpoints.....	47
Display.....	48
General.....	48
Snapshot File Dialogs.....	48
Emulation Control.....	48
Messages.....	49
Spooling.....	49
Library.....	49
Advanced.....	50
Logging.....	51
Logging.....	51
Diagnostic Logging.....	51
Snippets Menu.....	52
Snippet Editor.....	52
RZX Studio.....	53
Creating A New RZX Recording.....	53
Importing An Existing RZX Recording.....	54
Tape Browser.....	55
Loading A Tape.....	55
Creating A New Tape.....	57
ZX Spectrum.....	57
ZX80.....	58
ZX81.....	58
Jupiter ACE.....	58
Microdrives.....	59
Microdrive Map.....	61
Sector List.....	63
Disks.....	64
ZX Printer.....	67

Debugger.....	67
Toolbar.....	68
Edit Menu.....	69
View Menu.....	69
Debug Menu.....	70
Disassembly Area.....	70
Call Stack Area.....	72
Breakpoints Area.....	72
Expression Box.....	72
ROM and RAM Selectors.....	76
Disassembler.....	76
Memory.....	76
Tracking Memory Changes.....	78
Find Memory.....	79
Find.....	79
Find and Replace.....	80
Write.....	81
Z80 Registers.....	81
System Variables.....	82
Watches.....	83
Messages Window.....	85
Calculator.....	86
Assembler.....	86
Syntax.....	87
Directives.....	87
=.....	87
ALIGN n.....	87
ASSERT [expr].....	88
ASSERTIFDEF identifier.....	88
ASSERTIFNDEF identifier.....	88
ASSERTSTR S1,S2.....	88
BANK bank-num.....	88
BREAK [type] [,parameters...].....	88
DEFINE name value.....	91
DEFL name value.....	91
DEFNM msg.....	92
DEVICE target-device.....	92
DISPLAY msg.....	92
ELSE.....	92
END [start-address].....	92
ENDIF.....	93
ENDM.....	93
ENDPOKTRAINER.....	93
ENDR.....	93
ENDS.....	93
EQU address[,bank].....	93
EXECUTE execution-mode.....	93
IF expr.....	93
IFDEF identifier.....	94
IFLABEL label.....	94
IFNDEF identifier.....	94
IFNLABEL label.....	94

IMPORT item.....	94
INCBIN filename.....	94
INCHEX filename [address].....	94
INCLUDE filename.....	95
LABELSLIST.....	95
MACRO name [parameters...].	95
MEMCOPYMODE mode.....	96
ORG address.....	98
POKTRAINER trainer-name.....	98
REPT repeat-count.....	98
SAVEBIN [filename].....	99
SAVEHEX [filename].....	99
SAVELISTING [filename].....	99
SAVEPOK [filename].....	99
SAVESNA.....	100
SAVESNAP filename [,start-address].....	100
SAVESYMBOLS filename.....	100
SAVETAP filename[,start-address].....	100
SAVETAPE filename[,loading-screen-filename].....	100
SETREGISTER register, value [,register, value...]	100
STRUCT name [,initial-offset   base-struct].....	101
TAPEINFO category:text.....	102
TAPELOADEREXEC command.....	103
TAPEPROGNAME name.....	103
TARGET machine.....	103
TRACE [OFF   ON].....	104
UNREFS.....	104
VERBOSE [OFF   ON].....	104
Tape Loader Generators.....	104
ZX80.....	104
ZX81.....	104
ZX Spectrum 16k, 48k.....	105
ZX Spectrum 128K and later.....	105
Scripting.....	105
Lua Extensions.....	107
print(args...).....	107
tprint(table).....	107
dirs (path[, {options}]).....	107
Status Codes.....	109
inks.add_breakpoint({parameters...}).....	110
inks.add_log_capture([filters]).....	111
inks.add_message_for_user(message[, message type][, show-time-secs]).....	112
inks.add_rzx_rollback().....	113
inks.amend_breakpoint(id=breakpoint_id,[parameters...]).....	113
inks.assemble(assembly[,parameters]).....	113
inks.assemble_file(file[,parameters]).....	115
inks.assembler_eval(expression).....	115
inks.attach_peripheral(peripheral[,peripheral]).....	115
Inks.check_mdrv_cart([unit-num]).....	116
inks.clear_captured_log_text(capture-id).....	116
inks.clear_ram([initialisation-value]).....	116
inks.compare_lines(lines1, lines2).....	116

inks.compress_basic_lines(lines).....	117
inks.crc32(string).....	117
inks.crc32_from_memory(address, length).....	117
inks.create_disk(drive-num, capacity).....	117
inks.create_machine(machine-name[,peripherals]).....	117
inks.create_mdrv_cart({ <i>config</i> },[ <i>Microdrive-unit-num</i> ]).....	118
inks.date_as_10_digit_string().....	119
inks.dbg_expand(string).....	119
inks.dbg_get_show_hex().....	119
inks.dbg_set_last_access(address, rw-value).....	119
inks.dbg_show_hex(show-hex-values).....	119
inks.default_profile_name().....	119
inks.delete_all_breakpoints().....	119
inks.delete_breakpoint(id).....	120
inks.delete_rzx_recording().....	120
inks.delete_user_messages().....	120
inks.detach_peripheral(peripheral[,peripheral]).....	120
inks.did_recording_play_successfully().....	120
inks.disassemble_to_file(out-file, start, end, [addresses[, generate-labels]]).....	120
inks.dofile(in-file).....	120
inks.dreamysleepynightiesnoozysnooze(time-to-sleep-for).....	121
inks.dump_profiles().....	121
inks.eject_disk(drive-num).....	121
inks.eject_mdrv_cart([unit-num]).....	121
inks.eject_tape().....	121
inks.enable_breakpoint(id[,enable]).....	121
inks.enable_canvas(enable-canvas).....	121
inks.enable_speech_subtitles([true false]).....	121
inks.enable_sysvar_symbols(enable).....	122
inks.eval(expr).....	122
inks.eval_watch(expr[,watch-format-specifier]).....	122
inks.expand_basic_command(command).....	122
inks.export_zx_printer(path).....	122
inks.file_extension(path).....	123
inks.file_name(path).....	123
inks.file_crc32(path).....	123
inks.file_sha1(path).....	123
inks.file_size(path).....	123
inks.file_stem(path).....	123
inks.files_in_library().....	123
inks.find_in_library(path).....	124
inks.flush_user_messages().....	124
inks.get_active_mdrv().....	124
inks.get_all_breakpoints().....	124
inks.get_all_model_names([long_names]).....	125
inks.get_all_model_nums().....	125
inks.get_attempted_load_file_class().....	126
inks.get_assembler_stats().....	126
inks.get_attached_peripherals().....	126
inks.get_base_folder().....	127
inks.get_basic_program([line-nums-as-keys[,show-hidden-numbers]]).....	127
inks.get_basic_program_from_tape(line-nums-as-keys).....	128

inks.get_basic_programs_from_mdrv(line-nums-as-keys[,unit-num]).....	128
inks.get_breakpoint(id).....	128
inks.get_breakpoint_ids().....	128
inks.get_captured_log_text(capture-id).....	129
inks.get_cwd().....	129
inks.get_disk_image_path(drive-num).....	129
inks.get_edition().....	129
inks.get_execution_state().....	130
inks.get_execution_state_str([execution_state]).....	130
inks.get_file_class(path).....	130
inks.get_file_class_str(file_class).....	131
inks.get_file_type(path).....	131
inks.get_frame_tstates().....	132
inks.get_full_address(addr-str   addr[,ram,rom]).....	132
inks.get_full_address_from_str(addr-str).....	133
inks.get_full_address_str(addr-str   addr[,ram,rom]).....	134
inks.get_last_breakpoint_hit().....	134
inks.get_last_spool_result().....	134
inks.get_machine_model().....	134
inks.get_machine_state().....	135
inks.get_mdrv_cart_image_path([unit-num]).....	135
inks.get_mdrv_cart_name([unit-num]).....	135
inks.get_model_name(model [,long_name]).....	135
inks.get_num_frames_executed().....	135
inks.get_num_rzx_rollbacks().....	135
inks.get_num_tape_blocks().....	135
inks.get_peripheral_features(peripheral).....	136
inks.get_ram_name(ram-page).....	136
inks.get_ram_pages().....	136
inks.get_recent(item-num[,type]).....	136
inks.get_recent_count([type]).....	137
inks.get_release_manifest().....	137
inks.get_release_version().....	137
inks.get_rom_name(rom).....	137
inks.get_rom_pages().....	137
inks.get_screen_as_text([mode]).....	137
inks.get_slr_levels().....	139
inks.get_snapshots_folder().....	139
inks.get_speed().....	139
inks.get_system_editor_cursor_mode().....	140
inks.get_total_tstates().....	140
inks.get_z80_register(reg-name[,reg-name...]).....	140
inks.has_disk_controller().....	140
inks.hex(number[,num-bits]) inks.hex({numbers}[,num-bits]) inks.hex(string).....	140
inks.insert_disk(drive-num, path).....	140
inks.insert_tape(path).....	140
inks.insert_mdrv_cart(cartridge-path[,read-only[,unit-num]]).....	141
inks.is_breakpoint(id).....	141
inks.is_canvas_enabled().....	141
inks.is_debug_build().....	141
inks.is_directory(path).....	141
inks.is_disk_inserted(drive-num).....	141

inks.is_disk_modified(drive-num).....	141
inks.is_drive_active(drive-num).....	141
inks.is_file(path).....	142
inks.is_keyboard_assist_active().....	142
inks.is_mdrv_cart_inserted([unit-num]).....	142
inks.is_mdrv_cart_modified([unit-num]).....	142
inks.is_mdrv_unit_present([unit-num]).....	142
inks.is_paused().....	142
inks.is_peripheral_attached(peripheral).....	142
inks.is_playing_recording().....	142
inks.is_playing_rzx().....	142
inks.is_ready_to_spool_line().....	142
inks.is_recording_audio().....	143
inks.is_recording_av().....	143
inks.is_recording_gif().....	143
inks.is_recording_rzx().....	143
inks.is_spooling().....	143
inks.is_supported_file_class(path[, {classes-to-include}, {classes-to-exclude}]).....	143
inks.is_supported_file_type(path[, {types-to-include}, {types-to-exclude}]).....	143
inks.is_system_error_code(err-code).....	144
inks.is_system_ready_for_keyboard_assist().....	144
inks.is_tape_present().....	144
inks.joystick_name(joystick).....	144
inks.keyboard_assist(command).....	144
inks.keyboard_assist_select_system_menu(menu-num).....	144
inks.line_count(text).....	144
inks.load(path).....	145
inks.load_as(path, as-extension).....	145
inks.load_file_data(path[, fuzz-data, fuzz-num]).....	145
inks.load_from_library([mark-as-selected[, file-num]]).....	145
inks.log(msg[, message-type]).....	145
inks.may_spool().....	145
inks.mdrv_cat([unit-num]).....	146
inks.path_from_library([mark-as-selected[, file-num]]).....	147
inks.pause(pause).....	147
inks.peek(address-string   address[, ram-page, rom]).....	147
inks.peek_a(address-string   address[, ram-page, rom]).....	147
inks.peripheral_feature_name(peripheral-feature).....	147
inks.peripheral_name(peripheral).....	148
inks.poke(address-string   address, value [, ram-page]).....	148
inks.poke_a(address-string   address, value [, ram-page]).....	148
inks.query_peripheral_feature(peripheral, peripheral-feature).....	148
inks.query_peripheral_feature_state(peripheral, peripheral-feature).....	148
inks.randomise_ram().....	149
inks.read_as(data, extension[, add-to-recent]).....	149
inks.read_mdrv_cart_file(path[, unit-num]).....	149
inks.relative_to_script(filename).....	149
inks.replace_extension(path, new-extension).....	149
inks.reset_machine([hard-reset]).....	150
inks.resume_recording_rzx([path]).....	150
inks.rollback_rzx_recording().....	150
inks.save(path [, add-to-recent]).....	150

inks.save_disk(drive-num).....	150
inks.save_disk_as(drive-num, path).....	150
inks.save_gif(path[,include-border[,animated]]).....	150
inks.save_mdrv_cart([unit-num]).....	150
inks.save_mdrv_cart_as(path[,unit-num]).....	151
inks.save_rzx_recording(path).....	151
inks.save_screen_as_text(path).....	151
inks.set_peripheral_feature_state().....	151
inks.set_profile(controller-type [,profile-name]).....	151
inks.set_profile_joystick(controller-type, emulated-joystick [,profile-name]).....	151
inks.set_speed(speed-percent).....	151
inks.set_z80_register(reg-name,value[,reg-name,value...]).....	152
inks.sha1(string).....	152
inks.sha1_from_memory(address, length).....	152
inks.sleep(time-to-sleep-for).....	152
inks.spectrum_screen_address(x,y).....	152
inks.spool(path   {spool-lines}[,allow-immediate-commands]).....	152
inks.start_recording_audio(path).....	152
inks.start_recording_av(path[,audio-bitrate][,video-bitrate],[include-border]).....	153
inks.start_recording_gif(path).....	153
inks.start_recording_rzx(path).....	153
inks.status_str(status-code).....	153
inks.still_here().....	153
inks.stop_keyboard_assist().....	153
inks.stop_playing_rzx().....	153
inks.stop_recording_audio().....	153
inks.stop_recording_av().....	153
inks.stop_recording_gif().....	153
inks.stop_recording_rzx().....	154
inks.stop_script([message[,message-style]]).....	154
inks.stop_spooling().....	154
inks.system_error_str(code).....	154
inks.sysvar(system-variable-name).....	154
inks.table_size(table).....	154
inks.tape_is_code_loader().....	154
inks.tape_play_from_block(block-num).....	154
inks.temp_folder().....	155
inks.throttle([throttle]).....	155
inks.to_base(number [,base[,number-of-bits-to-display]]).....	155
inks.tokenise_into_basic(BASIC-command).....	155
Inkspector Command Line Tool, incli.exe.....	155
Command Line Reference.....	156
--avcfg arg.....	156
--flashgif.....	157
--help.....	157
--hex [arg].....	157
--listbasic.....	157
--listbasicvars.....	158
--listconstants.....	158
--listdblog.....	158
--listdirectives.....	158
--listlua.....	158



--listmdrvcart [arg].....	158
--listsysvars [arg].....	159
--listtape.....	159
--load arg.....	159
--log.....	160
--machine arg.....	160
--noloadconfig.....	160
--peripheral arg.....	160
--playrecording.....	161
--playrzxfolder arg.....	161
--playtape.....	161
--postscript arg.....	161
--prescript arg.....	162
--quiet.....	162
--readonly.....	162
--recaudio arg.....	162
--recav arg.....	162
--recgif arg.....	162
--save arg.....	162
--savegif arg.....	162
--snippet arg.....	163
--state.....	163
--timeout arg.....	163
--verbose [arg].....	163
Reference.....	163
Expression Evaluator.....	163
Operators.....	163
Pseudo Variables.....	164
Z80 Registers.....	165
Source Code Support.....	166
Creating .lst and .sym files using TASM.....	166
Creating .lst and .sym files using SjAsmPlus.....	166
Creating .sym files using PasmO.....	166
Supported File Formats.....	166
Extensions to the .szx format.....	168

## Introduction

Inkspector is a program that assists you, the talented canine hair stylist, in prototyping, designing and previewing wigs for dogs.

If only.

No, I'm afraid this is just yet another program that emulates the Sinclair ZX Spectrum and related models, the Sinclair ZX80, Sinclair ZX81 and Jupiter Cantab's Jupiter ACE. You can play games with Inkspector, or hack them using its debugger. You can even use the built-in macro assembler to write new programs and games, or create .pok files to patch existing games.

You can write scripts to convert a folder full of .rzx recordings and snapshots to static or animated .gif files.

You can tell Inkspector where your hoard of ZX files are to allow you to quickly search through them, even when you have hundreds of thousands of them.

There's literally ~~two or three~~ one or two different things you can do with this miracle of software development. So, without further ado, let's see what this thing is capable of!

## Main Window

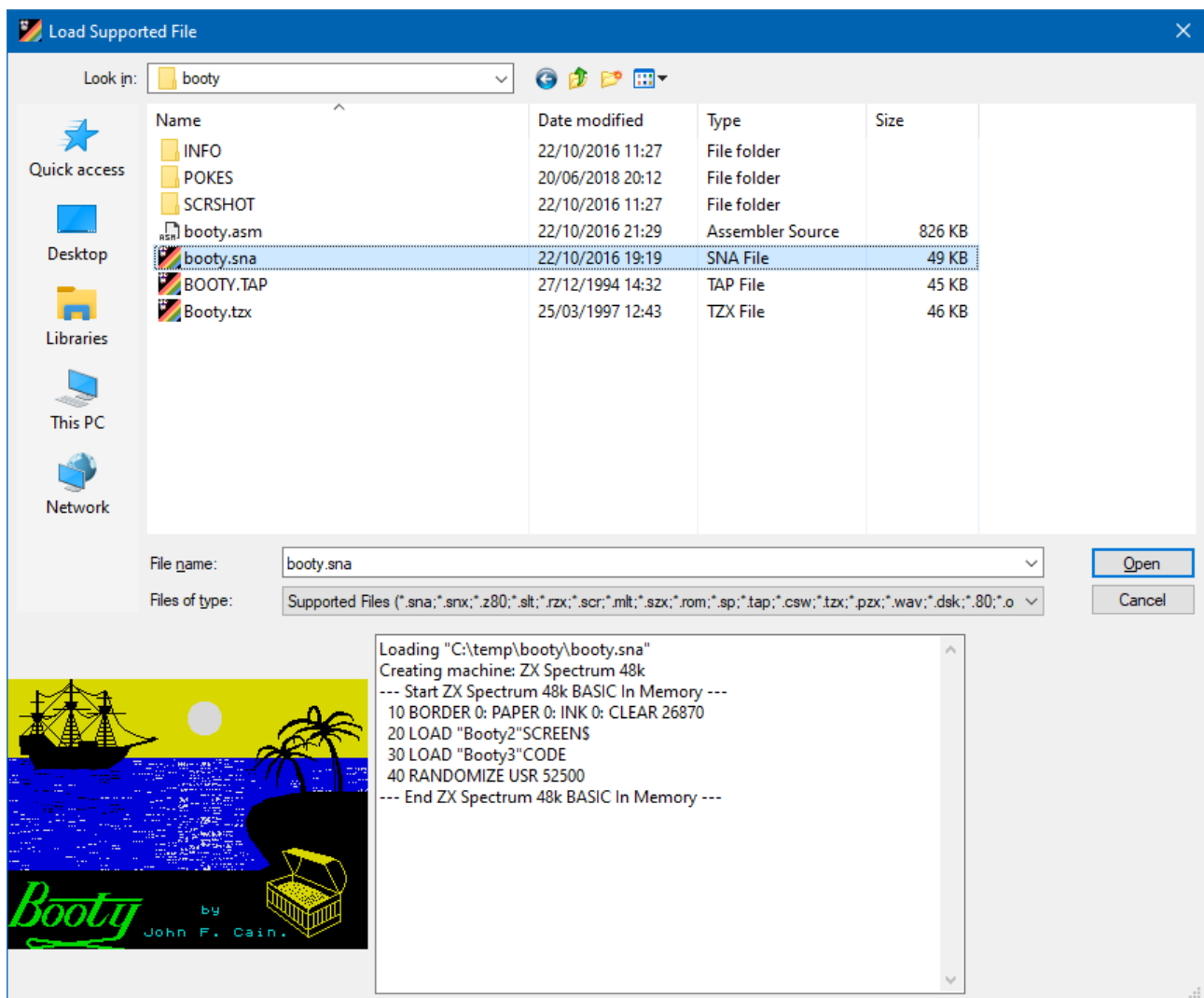
When Inkspector is started, you will be presented with the machine that was in use when Inkspector was last used, or a 48k Spectrum if it's the first time it has been run. You can [configure it](#) to start with a specific machine if you prefer.

Unless you're in a creative mood and ready to do some hard-core retro programming, you'll probably want to load an existing snapshot, .rzx recording or other type of [file](#). Well, you're in luck. We can do that.

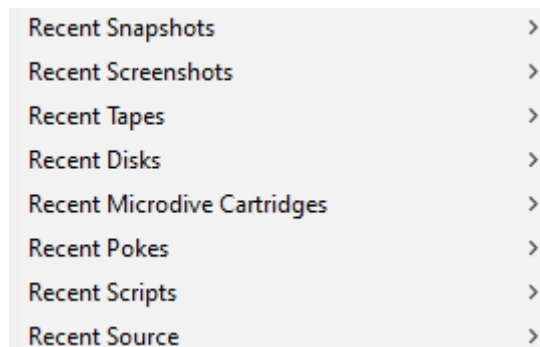
## File Menu

From the File menu select Load Supported File or click on the leftmost icon on the toolbar or press F3. Doing any of these things will bring up the Load Supported File dialog. This is like every other file selection dialog you've ever seen, except this one also shows you a live preview of the currently selected snapshot, along with some information about it. For example, when selecting booty.sna, the snapshot is previewed (without affecting the emulated machine in the main Inkspector window – nothing is loaded into the main window's machine until you double-click a file or press ENTER to select one).

Sadly I can't demonstrate in this document the beauty of Booty's shimmering water in the file preview screen-grab below, but you get the gist. The amount and type of information that's shown in the adjacent text window is configurable on the Options → General page, but more about that [later](#). So much time and so little to do. Wait a minute. Strike that. Reverse it.



If the selected file is loaded successfully, its name is filed away in one of the “Recent...” item categories on the File menu to allow it to be re-selected easily. Each category remembers the ten most recently loaded files of that type.



### InkTip

If you hold down the CTRL key as you select an item from a Recent menu item (including Recent menus on other windows, such as the [Tape Browser](#)) the file’s name is copied to the clipboard rather than it being loaded.

The Load Most Recent Snapshot, Load Most Recent Tape, Load Most Recent Source menu items load the file at the top of the Recent Snapshots, Recent Tapes and Recent Source lists respectively.

Load Last Auto-Save loads the most recent machine auto-save image (see Options → [Start-up and Shut Down](#) under the Shut Down section). If no auto-save image is available the menu item is greyed out.

**InkTip**

If you hold down the CTRL key as you select Load Last Auto-Save, the auto-save image will be saved to the current user's %TEMP% folder. Since this image is normally stored in the configuration database, this is the easiest way to access it as a separate file. Depending on the machine it was saved for, it will be called auto-save.szx (for Spectrums), auto-save.ace (for the Jupiter ACE) or auto-save.z81 (for the ZX80 and ZX81).

From this menu you can also save a snapshot or a native-format screenshot (e.g. .scr or .mlt) of the current machine using Save Snapshot. Note that Inkspector allows ZX80, ZX81 and the Jupiter ACE to save out their current displays as Spectrum format .scr files too.

“Save Screen Image” allows you to save the contents of the machine's display as a .bmp, .png or .gif image file.

**InkTip**

If you wish to save a screen image (.bmp, .png, .gif) without a border, select Border → No Border from the Display menu first, otherwise the full border will be included in the image.

The intriguingly-named “Save Screen Image (with flashing)” saves the machine's display as a .gif file, just as Save Screen Image does, with the difference that if the current machine supports flashing (i.e. is any ZX Spectrum model) and the display currently contains at least one flashing attribute, a 2-frame animated .gif is written out to preserve the flashing effect. This allows you save such things as the Manic Miner loading screen in all its animated funkiness.

**InkTip**

To control whether a border is included in the flashing screen image, select “Include the Spectrum's border in the output” on the Options → Recordings page in the Animated GIFs section.

“Save Screen As Text” attempts to save the contents of the machine's screen as an ASCII text file. I say attempts, as there are limitations – see [inks.get\\_screen\\_as\\_text](#) for more information.

The raw contents of the machine's memory can be saved as a file using Save Memory.

Files that are successfully saved are also filed away in the appropriate Recent menu. Note that the “Screenshots” category refers to .scr and .mlt ZX Spectrum screenshot files, not image files such as .png, etc.

## Edit Menu

This main contains two ways to copy the current machine's display to the clipboard. The first, Copy Display, copies the current machine's display to the clipboard, including any border it may have.

The second menu item Copy Display As Text attempts to convert the current display to ASCII text before copying to the clipboard as text.

### **InkTip**

If you hold down the CTRL key as you select Copy Display As Text, line numbers will be included

## Display Menu

The Display menu is where you can change the size of the main window (100%, 200%, 300% or 400% of the actual machine display's dimensions – shortcuts keys Alt-Shift-1, 2, 3 and 4). You can also set a custom scale value anywhere between 100% and 400% (shortcut key Alt-Shift-5).

For machines that display a border area, you can select how much of it is displayed from the Border sub-menu.

You can also toggle Inkspector's full screen display from this menu as well as on the main window toolbar.

## View Menu

Raise All Windows brings all of Inkspector's tool windows to the top, making them visible again if they've become buried under other windows on the desktop.

Inkspector can display messages over the top of the machine's display area, for example, informing of the end of an RZX recording playback without being too disruptive. If the message has appeared at an inopportune moment, you can dismiss the message (and any more that are queued up) by selecting the Dismiss Information Bar menu item. Clicking the 'X' close button on the message itself closes the current message and immediately displays any other message that may be queued up.

The remaining menu entries are used to open up one of the tool windows that remain open in the background.

## Machine Menu

Playing with the default 48K ZX Spectrum machine is all very well, but there comes a time in everyone's life when they want a change. They might want to play with a 128K Spectrum. Or maybe tinker with some Forth using a Jupiter ACE. The Machine menu is where the machine can be changed, have its speed altered and peripherals added or removed on the fly.

As with the rest of this manual, I won't document menu items whose actions are obvious and don't have any hidden or unexpected behaviours. So on the Machine menu, Mute, Pause, Set Speed (which brings up the [General](#) options page where it is set) and Throttle all do as you'd expect.

Continuing with the Machine menu, you can add a peripheral without rebooting the machine by selecting a peripheral from the “Hot Peripheral Patching” sub-menu (incidentally, this does *not* mean sexy peripherals patching, but that peripherals may be attached or removed without rebooting the current machine). Only Peripherals supported by the current machine are listed. Note that peripherals added to the machine this way are only attached temporarily; Hard Resetting the machine, changing to another machine and back to the current model or restarting Inkspector will not re-attach a peripheral added this way. To add a peripheral to a machine permanently, visit the [Peripherals](#) options page, where you can also attach peripherals to specific snapshot files (“ooh, the mind boggles” I hear you say).

Soft Reset sets the Z80’s PC register to 0 to allow the emulated system to reset itself. This normally works fine as long as the machine itself isn’t in some bad state, such as having frozen or crashed in some spectacular way. If Soft Reset doesn’t work, you can always use Hard Reset which is akin to unplugging then plugging the power back in and will always reset the machine.

Generate An NMI causes the Z80 to generate a non-maskable interrupt. What this achieved depends on the machine currently running and the peripherals attached. On an unexpanded Spectrum it performs a soft reset in response, due to a bug in the [ROM’s NMI handler](#).

## Tools Menu

### Search Library

Because of the sheer the amount of software produced for the Sinclair machines, it can be difficult to quickly pinpoint a specific game or application amongst the thousands and thousands files available when they’re all sat in a big folder on your hard drive.

If you [configure Inkspector](#) with the location of one or more folders containing files [supported by Inkspector](#) this tool will display a filtered or complete list of them.

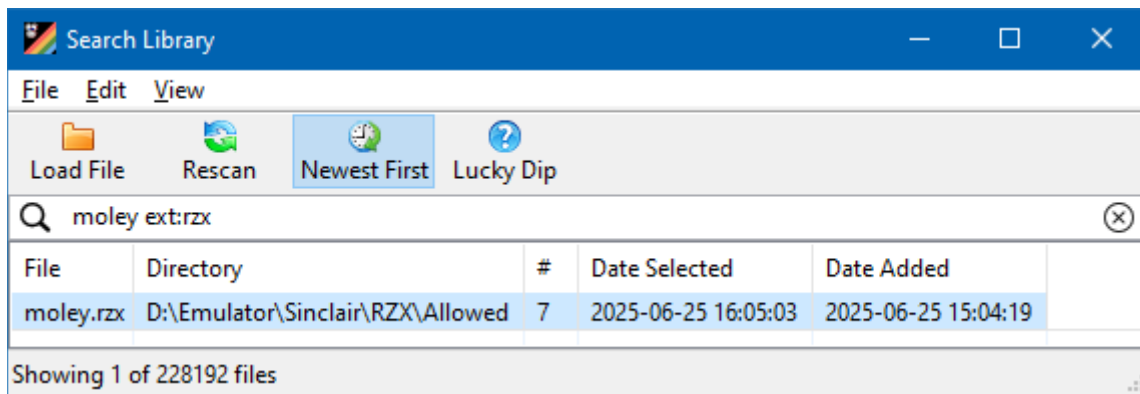
Opening this window displays the list of files matching the last filter used. Inkspector tries to be helpful and generally remembers things you’ve typed in before.

On my PC currently with no filter typed in, Inkspector Search shows “Showing 84480 of 84480 files”. Hmm...not particularly useful unless I fancy a lucky dip, scrolling down the list of files and picking a file at random. Funnily enough this is exactly what the Lucky Dip button does.

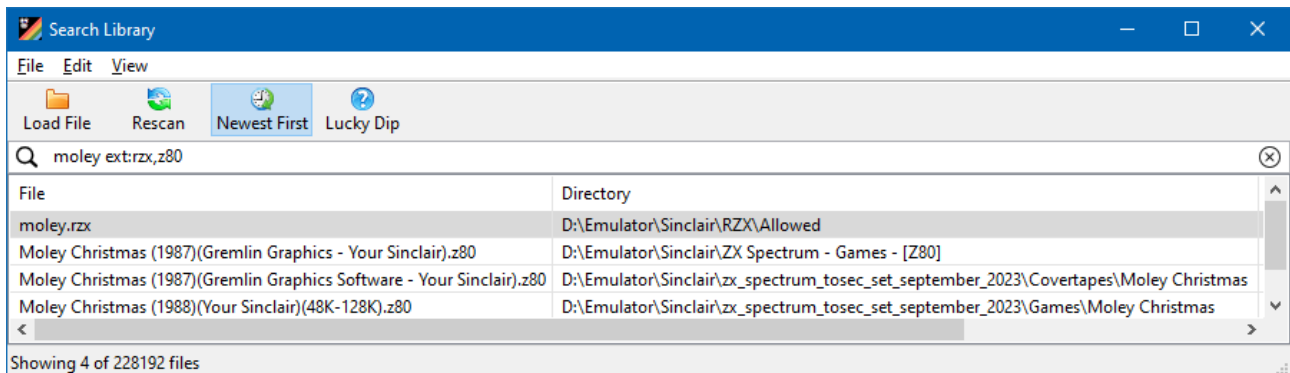
What if I want to play back an .rzx recording of Moley Christmas that I know is sat on my hard drive in my huuuuge Spectrum folder *somewhere*?

Typing “moley” (without the quotes of course) into the search box reduces the number of matching files down to about a dozen. Much, much better, and that’s enough for me to see the file I want. But we can do even better. Including ‘ext:extension[.extension2]’ in the search box filters the results down the file extensions supplied.

For example, typing “moley ext:rzx” shows just one file in the results list! Hurrah! If I double-click the file, it is loaded up in the main window and Inkspector records the current time for moley.rzx’s “Date Selected” column, as well as increasing the number of times the file has been selected from this window, shown in the ‘#’ column.



If I add “.z80” to the ext: filter to include .z80 files too, I get the following:



If the file you’re searching for doesn’t appear in the search results when you expect it to, you may need to perform a rescan by pressing the Rescan button to ensure it’s included in the list of files Inkspector knows about (because this can be a time-consuming operation, it isn’t done by default, but can be [configured to do so](#)). Personally, I find the occasional manual Rescan is enough as my collection is reasonably static.

### InkTip

Holding down the CTRL key while selecting a file to load will close down the Search window if the file loads successfully.

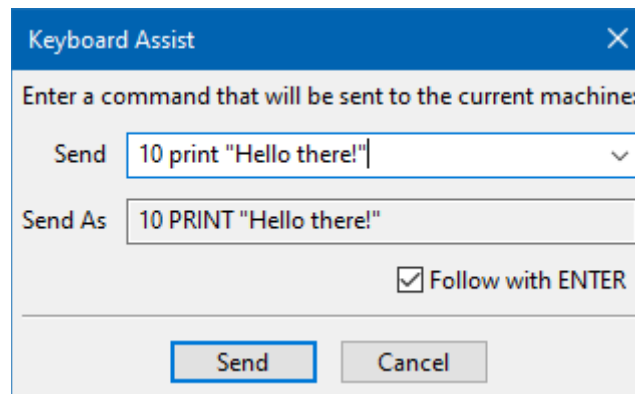
## Keyboard Assist

Even with a picture of a Sinclair machine’s keyboard to hand, entering BASIC commands can sometimes be tedious, especially when extended modes and *shifted* extended modes are involved. Keyboard Assist allows you to enter a BASIC (or Forth, for the ACE) command in Inkspector and it will do the typing for you, saving you from getting your phalanges knotted.

As a simple example, with a freshly booted up Spectrum 48K, press Alt-K or press the Key Assist toolbar button (the magic wand) to open the Keyboard Assist dialog and type in the following in the ‘Send’ box:

```
10 print "Hello there!"
```

The dialog will now look as follows. Notice how “print” has become uppercase in the ‘Send As’ preview, acknowledging it is a BASIC keyword.



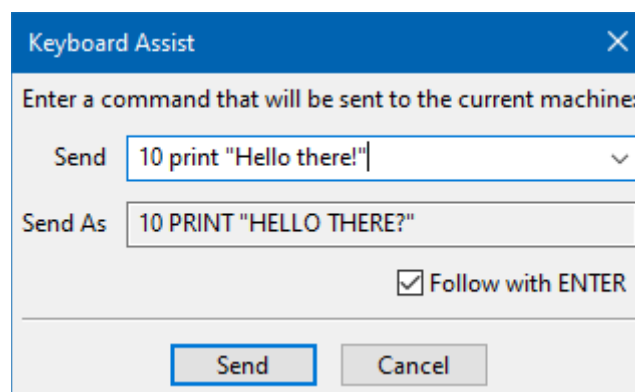
Press “Send” and you will be rewarded with the following being typed eerily:

```
10>PRINT "Hello there!"
```

Keyboard Assist works for all the machines emulated by Inkspector, but sometimes substitutions are made in the keys actually pressed to work around limitations of the underlying machine. For example, using the Machine → Machine menu, select a ZX80 and then bring up Keyboard Assist again and type the exact same again:

```
10 print "Hello there!"
```

Note how this time, not only has PRINT been converted to uppercase, but the text inside the quotes too. This is because the ZX80’s character set doesn’t contain lowercase characters (among many other omissions). Neither does it contain an exclamation mark, which is why it’s been substituted with a question mark here.

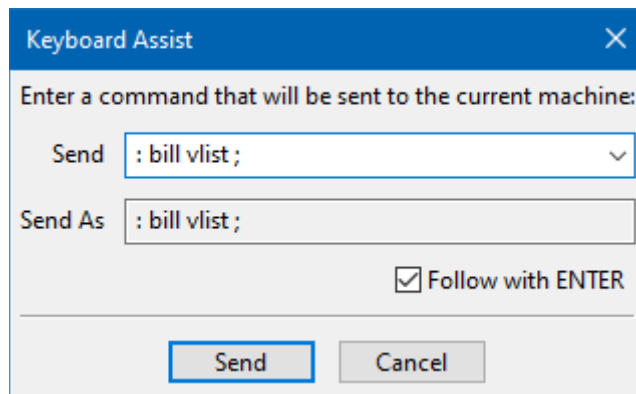


And when we press “Send” the command is typed in exactly as previewed:

```
10>PRINT "HELLO THERE?"
```

Even on systems that don’t use a keyword entry system, such as the Jupiter ACE or the Spectrum 128 and later, it can still be a useful tool to avoid having to experiment which keys need to be shifted to produce semicolons and other characters that require shifted keypresses. For example on the Jupiter ACE:



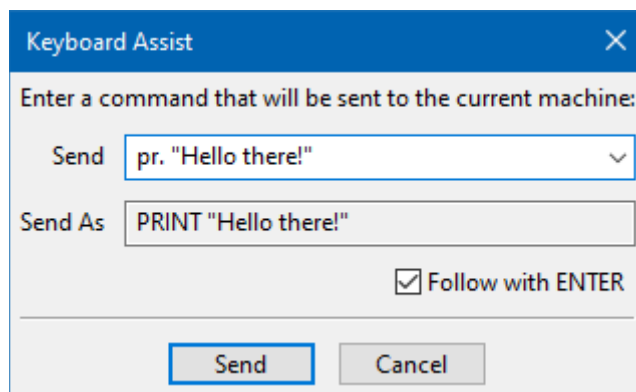


```
: bill vlist ; OK
```

Note that Keyboard Assist maintains a separate “Send” history for each machine.

### **InkTip**

Keywords may be specified in an abbreviated form by using a period, e.g. 10 pr. "Hello there!". Where the abbreviation is ambiguous (e.g. “p.” which could be one of several keywords) the keyword is not expanded.



## **Add Breakpoint**

First of all, it may seem a little odd that this window is accessible from the main window, outside of the debugger (it is accessible from there too of course) but I was finding myself setting breakpoints without needing to open the debugger first, when already knowing an address to break on or port being read from. So I added this shortcut. When a breakpoint is hit, the debugger will open automatically if it's not already.

Let's start with a lovely picture of the breakpoint window.

×

Add Breakpoint

Breakpoint

Breakpoint Type

Op-code Read

Address

\$0000

Symbol

Address Mask (0 to match all)

\$FFFF

ROM / RAM Page

Any

Condition

Run Snippet

None

Break When

Always break

1

Comment

Current Hit Count: 0

Reset

☐ One-Shot

☒ Enabled

OK

Cancel

**Breakpoint Type** declares the type of breakpoint to be added. It may be one of the following:

Type	Breaks When
Op-code Read	The Z80 performs an op-code read (i.e. reads the first byte of an instruction).
Memory Read	The Z80 performs a memory read
Memory Write	The Z80 performs a memory write
Memory Read/Write	The Z80 performs a memory read or write
Port Read	The Z80 performs a port read
Port Write	The Z80 performs a port write
Port Read/Write	The Z80 performs a port read or write
Maskable Interrupt	A maskable interrupt is taken
DI/HALT	The Z80 executes a HALT instruction when maskable interrupts are disabled
Condition	The expression in the Condition box evaluates to true (non-zero) regardless of when it occurs. i.e. it doesn't have to be triggered by a memory or port access. It is evaluated at the start of every Z80 instruction.
Re-triggered Interrupt	The Z80 executes a second maskable interrupt for the same instance of the INT line being held low by the ULA.
NMI Interrupt	A non-maskable interrupt is taken

**Address** specifies the memory or port address for the breakpoint.

**Address Mask** is applied to the **Address**. If the mask value is zero, it means all addresses will be considered (effectively the **Address** value is ignored). If the mask value is non-zero and the breakpoint address (port or memory) is equal to *the current memory or port address being accessed ANDed with the address mask value*, the breakpoint is considered. One use of the **Address Mask** is for trapping partially decoded addresses, such as accesses of the Spectrum's ULA. The usual convention for programmers is to access the ULA using port 254, but due to the partial address decoding, any port with the bottom bit reset will access it, so breakpointing on Port 254 might not catch all ULA accesses. However, setting the breakpoint's **Address** to 0 (because we're only interested in the bottom bit and only when it's reset) and the **Address Mask** to 1 (again, because we're only interested in the address's bottom bit) will catch all ULA accesses, even if, for example, port \$7E is used.

**ROM / RAM Pages** allow the breakpoint to be set for a specific page, for example if wishing to set a breakpoint when an op code is read in the ZX Interface 1 ROM, not in the standard ZX Spectrum ROM, or a memory read from the 128K's RAM page #4, etc.

**Condition** contains an expression to be evaluated if all the other breakpoint conditions are true. If it's empty, or the expression evaluates to true, the breakpoint is considered. See the [Expression Evaluator](#) reference for the full list of what's available. For Condition breakpoints the condition cannot be empty.

Who's for some examples?

Example Condition	Breaks If All Other Conditions Are True When...
@addr >= 49152 && @addr <= 49160	The memory or port address (depending on the breakpoint type) is within the range 49152 and 49160 inclusive
pc > \$4000	The Z80's PC register holds a value greater than \$4000 hex (16384 decimal)
*\$8000==\$1234	The memory at address \$8000 (low byte) is \$34, and at \$8001 (the high byte) is \$12
LOW *\$8000==\$34	The byte at memory address \$8000 is \$34
de > hl && de < bc	The Z80 register de is greater than the value of hl and less than the value of bc
@nz	The Z80's Z flag is reset (not zero)
@m	The Z80's sign flag is set
*curSpr==FirstSprite	The 16-bit pointer at address curSpr (presumably a symbol from a snapshot or assembly) is pointing to another symbol called FirstSprite
hl==16384	The value of the HL register pair is 16384
hl'==16384	The value of the HL' (alternate) register pair is 16384

**Run Snippet** allows one of the ten snippets to be executed when the breakpoint hits.

**Break When** determines when the breakpoint stops execution if all the conditions meet.

Break When...	Allows The Breakpoint To Hit When
Always Break	Always breaks. This is a typical breakpoint.
Break When Equal To	Breaks when the hit count is greater than the value specified
Break When Equal Or Greater To	Breaks when the hit count is greater or equal to the value specified
Break When Multiple Of	Breaks when the hit count is a multiple of the value specified
Never Break	Never breaks. Although this might sound odd, it can be useful if only wanting to run a snippet then continuing execution.

**Comment** is a free text field that is shown when breakpoints are referenced. It is not used by the debugger and has no effect on when the breakpoint hits. It may be empty.

**One-Shot.** When checked, the breakpoint is a one-shot breakpoint that deletes itself the first time it is hit.

**Enabled** determines whether the breakpoint is active or not. Enabled breakpoints are shown as solid red circles in the debugger and red boxes in the memory window (as per the picture below). Disabled breakpoints are shown as hollow red circles in the debugger and grey boxes in the memory window.

To modify or delete an existing breakpoint, double click one shown in the debugger's breakpoint list.

Certain types of breakpoint can be created from the memory window by right-clicking on a byte, e.g. Breakpoint on Execute. Addresses that have breakpoints attached to them are shown in red. For example, a Breakpoint on Execute at address \$0000 looks like this:

Address: \$0000, 48K ROM = \$F3 (243), Breakpoint [2] Op-Code Read  
\$0000 **F3** AF 11 FF FF C3 CB 11 2A 5D 5C 22 5F 5C 18 43 C3 F2 15 FF FF FF FF 2A **F3** .....\*]

## Poke Memory

**Poke Memory**

Address: 0xFFE7 Symbol

Mnemonic: ld a,':push bc Assemble

Value: 0x0002 Poke

Hex. Output: 02 Poke

☒ Auto Increment ☒ Hexadecimal

OK

This is a simple memory pocker and assembler. The destination address may be selected from a symbol (such as the names of the system variables for the current machine, or ones imported from .sym files – see the [Debugger](#) options page) by pressing the Symbol button.

As well as being able to poke straight values, you may click on the “Assemble” radio button and enter one or more assembly instructions separated with a colon.

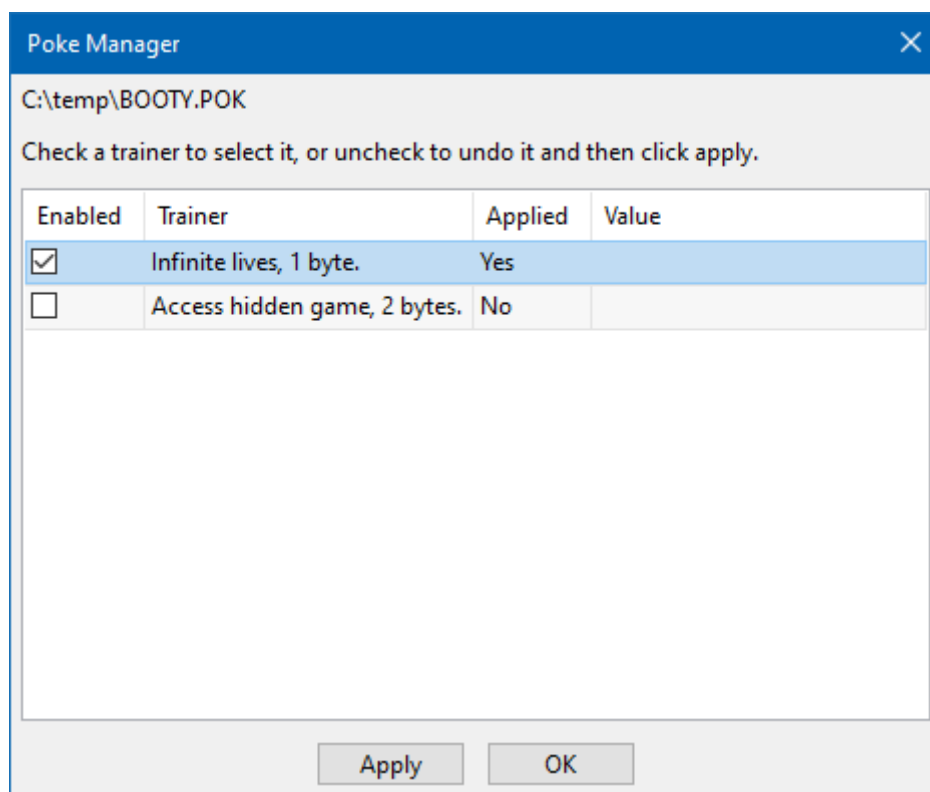
The Hex. Output box shows the bytes that will be output to the current address when the Poke button is pressed.

### **InkTip**

Changing the Hexadecimal option takes immediate effect not only on this window, but on all open Inkspector windows where numbers are displayed. The same is true for any Inkspector window where you can switch between hexadecimal and decimal.

## **Poke Manager**

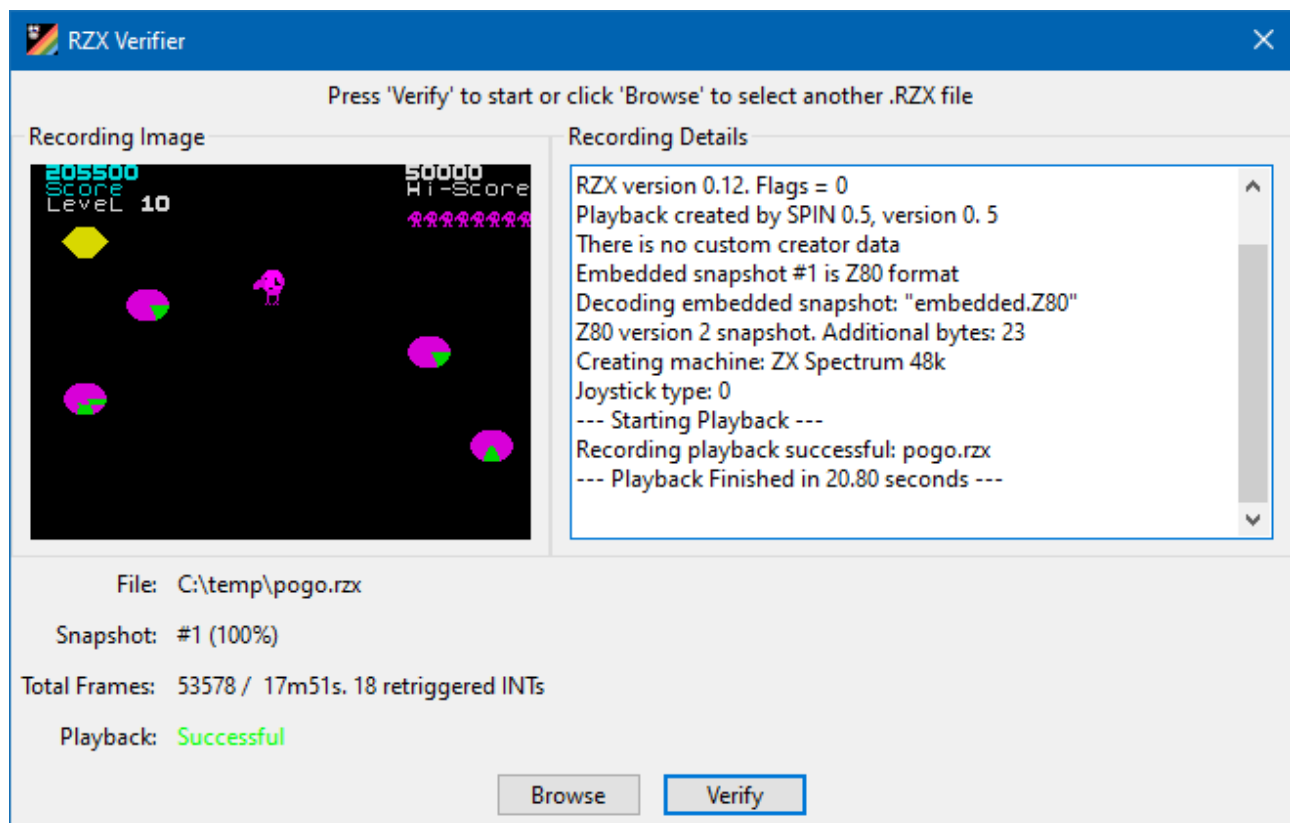
The menu item for Poke Manager is only enabled after a .pok file has been loaded. To enable or disable one of the trainers, check or uncheck them and then press Apply. The “Applied” column should reflect the “Enabled” column after pressing Apply.



## **RZX Verifier**

The RZX Verifier is a tool I hadn't intended to keep. It was written to help me when I added RZX recording, but I didn't have the heart to sack it once it was done, so here it is. It's used to verify that an .rzx recording file plays successfully to completion. The playback is performed as quickly as possible, similarly to how it would be if played on the main window with Throttle disabled. If

playback fails before reaching the end of the recording, details are shown in the Recording Details area, and of course the [message window](#) if it's open.



## View BASIC Program

For all Sinclair machines, this tool displays in the message window the contents of any BASIC program currently present in the system. The message window is opened for you if not already open. If we were to run this after spooling the [example](#), we would see:

```
10 BORDER 0[0]: PAPER 0[0]: INK 7[7]: CLS : REM black out screen
20 LET x1=0[0]: LET y1=0[0]: REM start of line
30 LET c=1[1]: REM for ink colour, starting blue
40 LET x2=INT (RND*256[256]): LET y2=INT (RND*176[176]): REM random finish of
line
50 DRAW INK c;x2-x1,y2-y1
60 LET x1=x2: LET y1=y2: REM next line starts where last one finished
70 LET c=c+1[1]: IF c=8[8] THEN LET c=1[1]: REM new colour
80 GO TO 40[40]
```

Note how the numbers are immediately followed by another in square brackets. These are the actual representations of the number preceding them that BASIC uses, and which the systems hide when LISTing a program.

This tool can be useful disclosing a form of loading protection where the visible representation of a number has been changed from the actual (hidden) representation – the value in square brackets. If we load up Megadodo's Phoenix and run View BASIC Program we can see this being used:

```
1 CLEAR 24500[24500]
2 INK 0[0]: PAPER 0[0]: BORDER 0[0]: CLS
```

```

3 PRINT AT 8[8],7[7];"\016\005\019\001PHEENIX is loading";AT
10[10],9[9];"\019\000\016\006\017\002\018\001 Please Wait
\018\000\017\007\016\000"
4 POKE 23659[23659],0[0]
5 LOAD ""CODE 24576[24532]
6 LOAD ""CODE : RANDOMIZE USR 30121[30105]

```

Note how the numbers on lines 1 to 4 all look as expected, with the ones in square brackets matching the ones preceding them (incidentally, the 3-digit numbers following the \s on line 3 are the control characters used to set ink and paper colours for the loading message and nothing is amiss there). But wait! What's this on line 5? It *looks* like the code is being loaded to address 24576, but we can see it's actually being loaded to address 24532. Similarly, to start the game it would appear the code at address 30121 is being executed. But if we were to do that, the machine would show a pretty screen wipe effect and then reset, courtesy of a cheeky 'RST 0' instruction at address 30124. Of course the actual start of the game code is at 30105 as shown within the square brackets. Nice try Megadodo.

The ability to show the hidden values in square brackets can be turned off in the [Advanced options](#) page. It can also be changed to show the hidden value only when it is different from the visible value (i.e. when an attempt to thwart us has been made!). In which case lines 4 to 6 look like:

```

4 POKE 23659,0
5 LOAD ""CODE 24576[24532]
6 LOAD ""CODE : RANDOMIZE USR 30121[30105]

```

For the Jupiter ACE (which doesn't have BASIC of course), the list of currently defined Forth words is displayed, similar to how its own VLIST command works, except they're grouped either in the ROM or RAM section, depending on where the word is defined.

## View User Variables

This tool displays in the message window all BASIC variables currently defined in the active machine (except on the Jupiter ACE where it will just display "unsupported").

If you were to spool or type this program in to a Spectrum

```

10 LET a=1
20 FOR b=1 TO 10
30 FOR c=1 TO 100 STEP 10
run

```

Then press Alt-U (or select Tools → View User Variables) you will see:

```

a=1
b=FOR 1 TO 10 STEP 1 @ line 20:2
c=FOR 1 TO 100 STEP 10 @ line 30:2

```

Note how the FOR variables also show the line and statement within the line where a subsequent NEXT would continue execution.

All types of variables are shown, as can be demonstrated by spooling the following then pressing Alt-U. I've not included the output here as it's quite lengthy, due to all elements of the h dimension variable being shown. It'll be a nice surprise for you.

```

10 LET a=1

```

```

20 LET b=-20
30 LET c=1.23
40 LET d=-0.123
50 DIM e(10)
60 DIM a$(5,10)
70 DIM b$(10)
80 LET longName=42
90 FOR n=1 TO 20
100 DIM f(3,6)
110 LET g=-65535
120 LET h=-65536
130 FOR j=1 TO 100 STEP 5
140 DIM h(2,2,2,2,2,2,2,2,2,2)
150 LET n$="Marky"
run

```

Note that unlike the ZX Spectrum, the ZX81 does not have an integer number format, so if you were to spool the above program and press Alt-U, the value of A would be shown as 1.000000 and not 1.

## View Machine State

Clicking the Tool → View Machine State menu item or pressing Alt-A displays the state of the current machine and all its attached peripherals in the message window, opening it, if not already open.

Most of the information is self-descriptive. The Active Address Mapping section shows how the Z80's 64K address space is mapped to memory at that moment in time.

## Spool Clipboard, Spool File

Spooling allows one or more lines of commands to be typed in using the [Keyboard Assist](#) system. The commands may be in an ASCII text file or in the Windows clipboard as text. Spooling is started by selecting Tools → Spool Clipboard or Tools → Spool File.

If the [message window](#) is open, each line being spooled is echoed to it.

I find this feature particularly useful for trying out little BASIC code snippets that are often posted on Sinclair forums. Feel free to copy the block of BASIC below, taken from the Spectrum manual, to the clipboard then use Inkspector's Spool Clipboard function to enter it into the Spectrum.

```

10 BORDER 0: PAPER 0: INK 7: CLS: REM black out screen
20 LET x1=0: LET y1=0: REM start of line
30 LET c=1: REM for ink colour, starting blue
40 LET x2=INT (RND*256): LET y2=INT (RND*176):REM random finish of line
50 DRAW INK c;x2-x1,y2-y1
60 LET x1=x2: LET y1=y2: REM next line starts where last one finished
70 LET c=c+1: IF c=8 THEN LET c=1: REM new colour
80 GO TO 40
run

```

If you switch to a Jupiter ACE and attach the ETI Colour Board peripheral, spooling the following will show some pretty coloured bars.

```

16 base c!
: colourbars
2700 2400

```



```
do
i 4 / 8 mod
10 * 80 or
2700 c! 20 i c!
loop
87 2700 c!
0 2700 c!
;
colourbars
```

Note that any line starting with ‘#’ is ignored. To cancel any spooling operation already underway, select Tools → Stop Spooling.

## Assemble Clipboard Contents

This tool [assembles](#) the text contents of the clipboard.

## Recording

This sub-menu contains the items to start and stop the recording of audio, AV and GIF files.

### Record Audio

Select this to start recording to a .wav audio file. You will be prompted to enter the name of a .wav file to create. Recording continues until Inkspector exits, Record Audio is selected again to create another wav file, or Stop Recording Audio is selected. In all cases, the existing recording is stopped and finalised properly to ensure a usable .wav file is created.

### Record AV

Select this to start the recording of an .mp4 file containing video and audio. You will be prompted to enter the name of a .mp4 file to create. Recording continues until Inkspector exits, Record AV is selected again to create another mp4 file, or Stop Recording AV is selected. In all cases, the existing recording is stopped and finalised properly to ensure a usable .mp4 file is created.

### Record Animated GIF

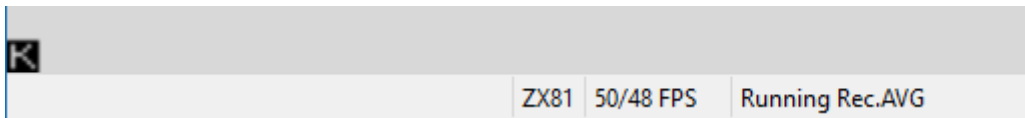
This tool starts the recording of an animated GIF from the current machine. When selected you will be prompted for the name of the .gif file to produce.

Recording continues until Inkspector exits, Record Animated GIF is selected again to create another gif file, or Stop Recording Animated GIF is selected. In all cases, the existing recording is stopped and finalised properly to ensure a usable .gif file is created.

There are [several options](#) available for GIF recording, such as the frame rate and whether to include a machine’s border area.

## Recording Indicator

Indication that one or more of the above recording methods is currently underway is shown on the status bar as “Rec.”, followed by A for audio, V for AV and G for GIF. For example, if we were to start all of the above recording types, we would see the following on the status bar:



## Stop Script

Since Inkspector scripts may continue running in the background once they've started (incidentally, [Snippets](#) may not – this is the main difference between the two), this menu option forces any such script to stop.

## Save Configuration

Although the Inkspector configuration is saved automatically after visiting the Options screen and when Inkspector is closed, it can also be saved manually at any time by clicking on this menu item. I sometimes find this useful because it also ensures the current Inkspector window and tool window positions are saved as currently arranged, which is useful for me when working on Inkspector as I often terminate it without closing it down properly (poor Inkspector).

## Options

This brings up the Options screen as you'd expect. Or you can press F10 to achieve the same. As it's such a large topic it has its [own section](#) below.

## Options Screen

From here you can change a multitude of settings to your heart's content to get Inkspector (and [incli](#), which shares the same configuration settings) to behave just as you wish...mostly.

## Start-up and Shut Down

### Start-up

Checking “Load any automatically saved machine state...” causes Inkspector to attempt to load the automatically-saved machine state last saved when Inkspector closed down. This requires the Shut Down option below to be selected too. If this option is checked and there is no last machine state

available, or there was a problem loading it, the (greyed out) Start-Up Machine option will be used instead.

Start-up Machine specifies which machine should be selected when Inkspector starts up (or loading the automatically saved state failed). The default is 'Machine Last Used' which selects the machine that was running when Inkspector last closed down.

Checking "Show the messages window" automatically opens the [Messages window](#) on start-up. I find this window so useful that I felt it deserved its own option to do this, even though it's also covered by the option below...

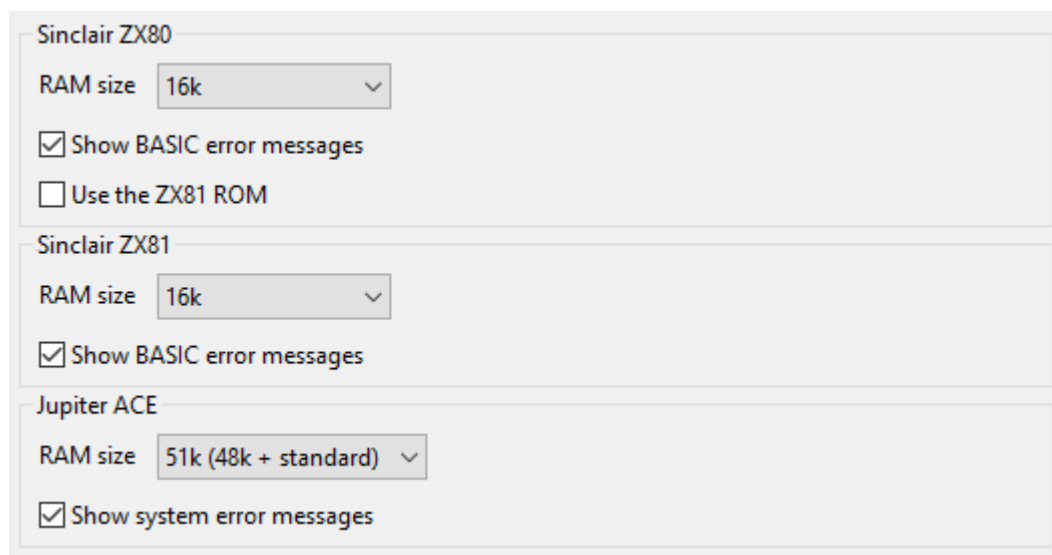
Checking "Automatically reopen active tool windows" opens all the tool windows (i.e. all the windows other than the main Inkspector window) that were open at the time Inkspector was last closed down.

"Run Snippet" allows you to select a snippet to be run on start-up once the machine has been selected as per the above options.

## Shut Down

Checking "Save the current machine state..." saves the state of the current machine when Inkspector closes down. This allows the "Load any automatically saved machine state..." option (above) to work, but can also serve as an automatic backup, in case you're a bit clumsy and have a habit of closing Inkspector down without first saving your work! In this case, you would run Inkspector again and select [File → Load Last Auto Save](#) to load it into Inkspector.

## ZX80, ZX81, Jupiter ACE



The screenshot shows a settings window with three sections for different machines:

- Sinclair ZX80**
  - RAM size: 16k (dropdown menu)
  - ☒ Show BASIC error messages
  - ☐ Use the ZX81 ROM
- Sinclair ZX81**
  - RAM size: 16k (dropdown menu)
  - ☒ Show BASIC error messages
- Jupiter ACE**
  - RAM size: 51k (48k + standard) (dropdown menu)
  - ☒ Show system error messages

On this screen are options to individually select the total RAM size of these three machines. The ZX80 has an additional optional to allow it to use the ZX81 ROM instead of the original ZX80 ROM. The ZX81 ROM was designed to be used as a drop-in replacement for the ZX80 ROM too, providing an improved BASIC with floating point numbers and the ability to use the ZX Printer.

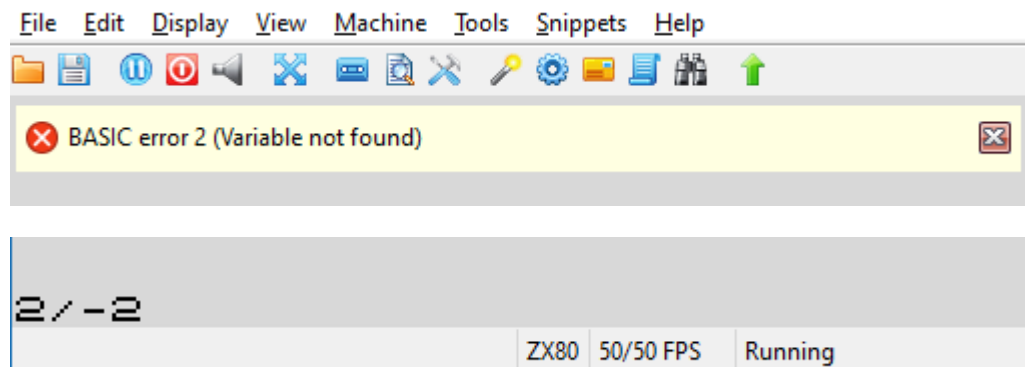
In addition, the "Show BASIC error messages" (for the ZX80 and ZX81) and "Show system error messages" (Jupiter ACE) can be checked, in which case a message will be displayed on the main

window when an error is encountered entering a BASIC or Forth command. This can be helpful as the native systems show a single digit code which is not particularly friendly.

For example, typing the following on a freshly booted ZX80

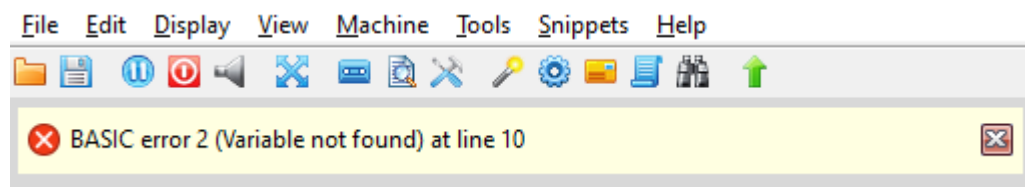
```
PRINT A
```

Results in a fairly cryptic “2/-2” being displayed in the BASIC entry area. With “Show system error messages” checked, you get the additional message shown for a few seconds at the top of the main window



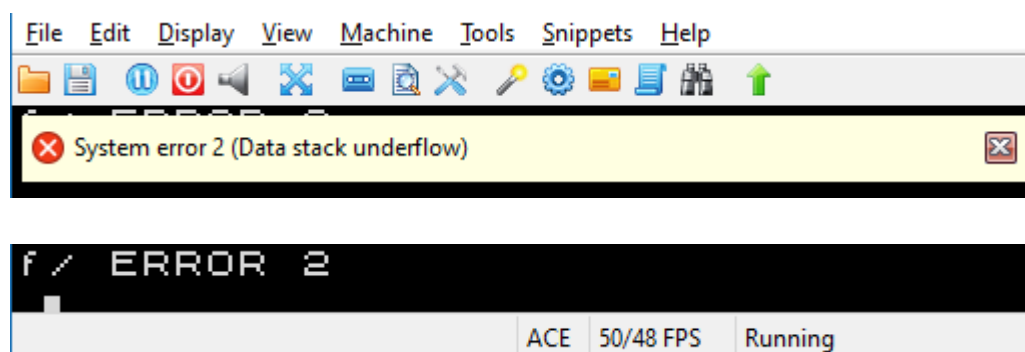
Or if the error occurs while running a program, the line number is also shown

```
10 PRINT A  
RUN
```



Or on the Jupiter ACE, after entering

```
f/
```



# ZX Spectrum

Timings

☐ Use late timings for the contention emulation

Spectrum 16K, 48K

Select which hardware issue to emulate

☐ Issue 2

☒ Issue 3

Select which Multiface model to use (automatic for other Spectrums)

☒ Multiface 1

☐ Multiface 128

☒ Multiface 1 is visible (uncheck for stealth mode)

Spectrum 128

☒ Substitute Spectrum 128 for a Pentagon 128K to allow more snapshots to load

Spectrum +3

☒ Enable drive B:

Kempston Joystick

☒ Emulate a genuine issue 4 Kempston joystick interface.  
Uncheck to emulate a clone or early issue Kempston.

## Timings

The display timings of some Spectrums start one Z80 t-state later than on others. Checking the “Use late timings...” box makes the emulated Spectrum timings start one t-state later too. .szx snapshots also contain a setting to control late timings and Inkspector honours this.

## Spectrum 16K, 48K

For 16K and 48K Spectrums, you can also select whether Issue 2 or 3 machines are emulated. This setting affects the value read from port 254 (Spectrum 128s and later models behave as issue 3). If an older game doesn’t appear to respond to the keyboard properly, it’s worth trying selecting Issue 2 and loading the game from tape. Note Inkspector uses the issue2/3 information in .szx and .z80 snapshots to set the issue when loading one, which is why the you would have to load the game from tape if you select issue2, as a snapshot might change the setting to issue 3).

Incidentally, the twitching movement of the player seen in early Imagine games (e.g. Ah Diddums!) is not due to the issue 2 or issue 3 emulation, but from the Imagine developers appearing to have developed those games with Fuller Boxes attached to their development machines, which they read from in their game code regardless, so when the games run on a machine that do not have a Fuller Box attached, the joystick value read from the (now absent) Fuller joystick part is supplied by the Spectrum’s erratic floating bus. You can prove this by [Hot Peripheral Attaching](#) a Fuller Box while playing Ah Diddums! which cures the teddy bear’s St. Vitus dance immediately.

In this section you can also select whether to use a Multiface 1 or Multiface 128 for the 16K or 48K Spectrum models. Additionally the Multiface 1 has a jumper setting that enables a stealth state, which may be emulated by unchecking the “Multiface 1 is visible...” box.

## Spectrum 128

When I originally tested Inkspector’s .rzx playback compatibility, I noticed there were a few .rzx files reported to be recorded using a Pentagon 128 (currently unsupported by Inkspector) that ran fine and played to completion when loaded into a Spectrum 128. Checking the “Substitute Spectrum 128 for a Pentagon 128K...” box will allow such .rzx files to load using a Spectrum 128 instead, which *might* allow the recording to play.

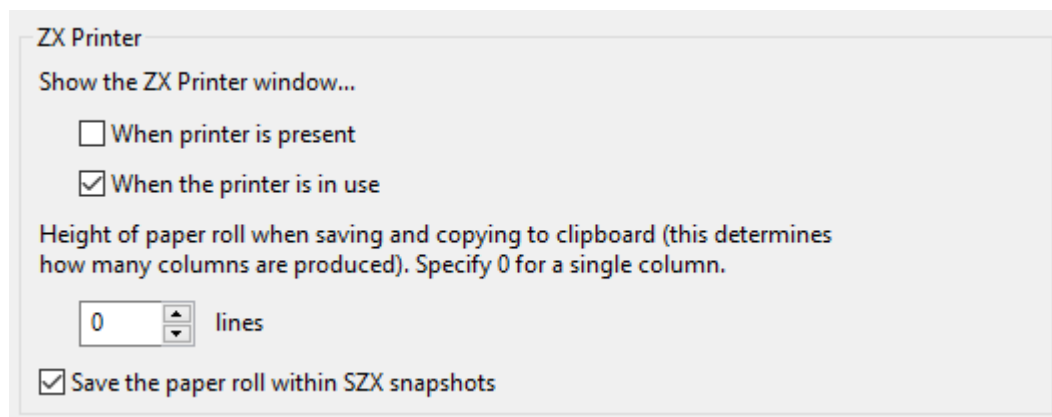
## Spectrum +3

Checking “Enable drive B:” enables a second floppy disk drive.

## Kempston Joystick

When the “Emulate a genuine issue 4 Kempston joystick interface” is checked, it can be read by reading any port between 0 and 31 (i.e. any address with the top three bits A7,A6,A5 all 0). When unchecked, the Kempston will respond to any port read with A5 reset.

## ZX Printer



The screenshot shows a settings window titled "ZX Printer". It contains the following options:

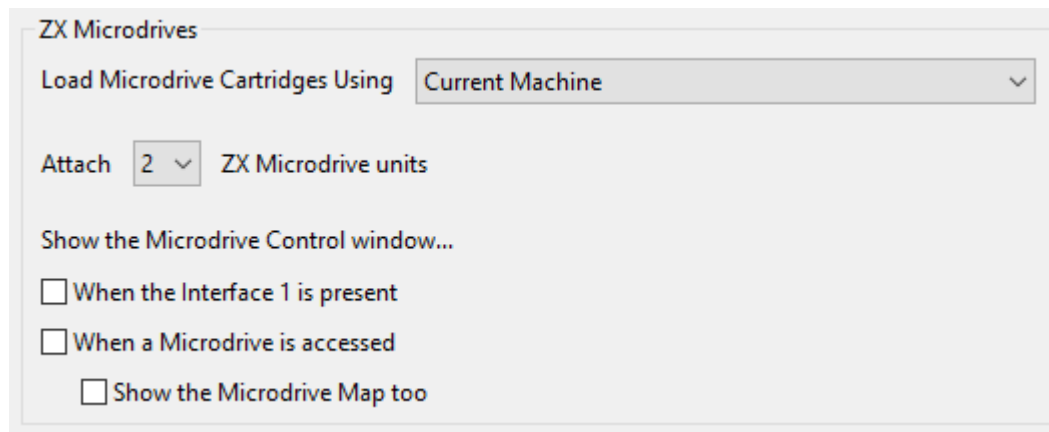
- A label "Show the ZX Printer window..." followed by two checkboxes:
  - ☐ When printer is present
  - ☒ When the printer is in use
- A text description: "Height of paper roll when saving and copying to clipboard (this determines how many columns are produced). Specify 0 for a single column."
- A numeric input field with the value "0" and a "lines" label.
- A checkbox ☒ Save the paper roll within SZX snapshots

The first two check boxes allow you to control whether the [ZX Printer](#) window opens automatically when the ZX Printer is either present and/or printed to.

The height of the printer roll to be used when saving to a picture image or copying to the clipboard, can be set here. The default of 0 lines means use a single column.

By checking the “Save the paper roll within SZX snapshots” box, Inkspector can add printer roll [information to .szx snapshot files](#) while remaining compatible with other emulators.

## ZX Interface 1



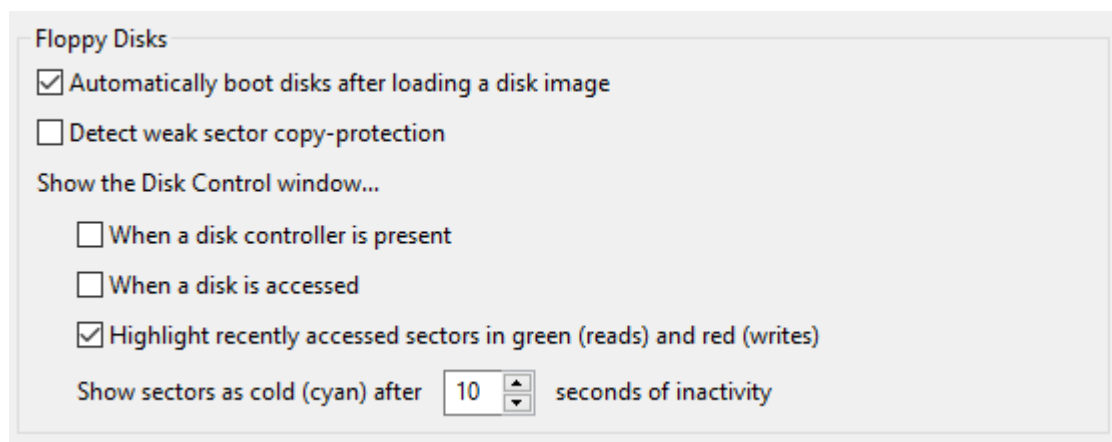
The screenshot shows the 'ZX Microdrives' configuration window. It has a title bar 'ZX Microdrives'. Below it is a dropdown menu 'Load Microdrive Cartridges Using' with 'Current Machine' selected. Below that is a label 'Attach' followed by a dropdown menu showing '2' and the text 'ZX Microdrive units'. Further down is the text 'Show the Microdrive Control window...' followed by three checkboxes: 'When the Interface 1 is present' (unchecked), 'When a Microdrive is accessed' (unchecked), and 'Show the Microdrive Map too' (unchecked).

When loading a .mdr Microdrive cartridge image file, the type of machine used to load it is specified with “Load Microdrive Cartridges Using”. ‘Current Machine’ will use the current machine, or a 48K Spectrum if the current model doesn’t support an Interface 1.

The ZX Interface 1 can support up to 8 Microdrive units, so you can specify between 0 and 8 here.

The remaining options allow you to control whether the [Microdrive Control](#) window opens (optionally with the [Microdrive Map](#) window too) when an Interface 1 is present or a Microdrive is accessed by a Spectrum.

## Floppy Disks



The screenshot shows the 'Floppy Disks' configuration window. It has a title bar 'Floppy Disks'. Below it are two checkboxes: 'Automatically boot disks after loading a disk image' (checked) and 'Detect weak sector copy-protection' (unchecked). Below that is the text 'Show the Disk Control window...' followed by three checkboxes: 'When a disk controller is present' (unchecked), 'When a disk is accessed' (unchecked), and 'Highlight recently accessed sectors in green (reads) and red (writes)' (checked). At the bottom is a label 'Show sectors as cold (cyan) after' followed by a spinner box showing '10' and the text 'seconds of inactivity'.

Checking “Automatically boot disks after loading a disk image” will take care of booting from the disk for you by rebooting the machine and automatically selecting “Loader” from the system menu once it has booted up.

“Detect weak sector copy-protection” allows some programs that use this type of copy protection to run when they otherwise wouldn’t.

“Show the Disk Control window...” options control whether the [Disk Control](#) window opens automatically when a floppy disk drive is present or accessed. Additionally the sectors most recently read or written shown on the [Disk Map](#) can be coloured for a short time to indicate the access pattern before they return to their regular (cold) colour of cyan. Green indicates a read and red indicates a write.

## Peripherals

Peripherals

Select which peripherals to attach to each machine

To add hardware customisations for a snapshot, press the Browse button

Machine or snapshot ZX Spectrum 48k ▼

Forget

Browse

Controller Profile <None> ▼

- ☐ Kempston Joystick Interface
- ☐ Kempston Mouse Interface
- ☐ Fuller Box
- ☐ Fuller Orator
- ☐ ZX Interface 1
- ☐ ZX Interface 2
- ☐ ZX Printer
- ☐ Currah MicroSpeech
- ☐ Currah MicroSource
- ☐ Didaktic Melodik
- ☐ Romantic Robot Multiface
- ☐ Cheetah SpecDrum

This page allows you to select which peripherals are attached, by default, to each of the machines emulated by InkSpector. Selecting a machine type from the drop down presents you with a list of peripherals supported by the selected machine. Checking the peripherals means that they will be attached to that particular machine whenever it is selected from within InkSpector.

If, after modifying the attached peripherals for the currently emulated machine, a reboot is required for the hardware changes to take effect, you will be prompted to reboot. For example, adding or removing a Currah MicroSpeech will always prompt as follows:

InkSpector Options

Do you wish to reset the machine for changes to take effect?

- Peripheral configuration changed

Yes No

As well as being able to select which peripherals are attached to a specific machine, you can also specify peripherals to attach when a specific snapshot or .rZX recording file has been successfully loaded. A good example of why you might want to do this is with the .rZX recording of Chuckie Egg that's available from all good Spectrum recording stockists. The game supports speech via the Fuller Orator speech box, but when the recording is loaded into InkSpector, the Orator isn't attached, because it isn't supported by the .z80 snapshot format that's embedded in the Chuckie Egg recording file and so InkSpector doesn't attach one (even if one has been specified to be attached to



a Spectrum machine in the peripherals screen – only peripherals supported by the loaded snapshot are normally ever attached to the machine, to prevent the snapshots from misbehaving by having unexpected peripherals present when they're run)

This means when playing back the recording we're losing out on hearing the lovely speech goodness that Chuckie Egg went to the trouble of adding back in the day. To work around this, Inkspector allows additional peripherals to be attached to specific snapshot and recording files. To do this, instead of selecting a machine from the first drop down box as you would normally, press the Browse button to identify the file you wish to have additional peripherals attached to (in our case, the Chuckie Egg recording). If we were to press Browse and then select the Chuckie Egg recording (let's assume the file is called chuckie.rzx), the peripherals page will then look like the following:

The screenshot shows the 'Peripherals' window in Inkspector. At the top, it says 'Select which peripherals to attach to each machine' and 'To add hardware customisations for a snapshot, press the Browse button'. The 'Machine or snapshot' dropdown menu is set to '<chuckie.rzx>'. Below this, the 'SHA-1' hash is displayed as '160651476ae451adc1671b8e024a80c097971a97'. To the right of the hash are 'Forget' and 'Browse' buttons. Below the hash, the machine model 'ZX Spectrum 48k' is shown, with another 'Browse' button to its right. The 'Controller Profile' dropdown is set to '<None>'. At the bottom, there is a list of peripherals with checkboxes: 'Kempston Joystick Interface' (unchecked), 'Kempston Mouse Interface' (unchecked), 'Fuller Box' (unchecked), 'Fuller Orator' (checked), and 'ZX Interface 1' (unchecked).

Note because we're attaching peripherals to a file (and not a machine type) the name of the file is shown within angled brackets ("<chuckie.rzx>") within the machine selection drop down box.

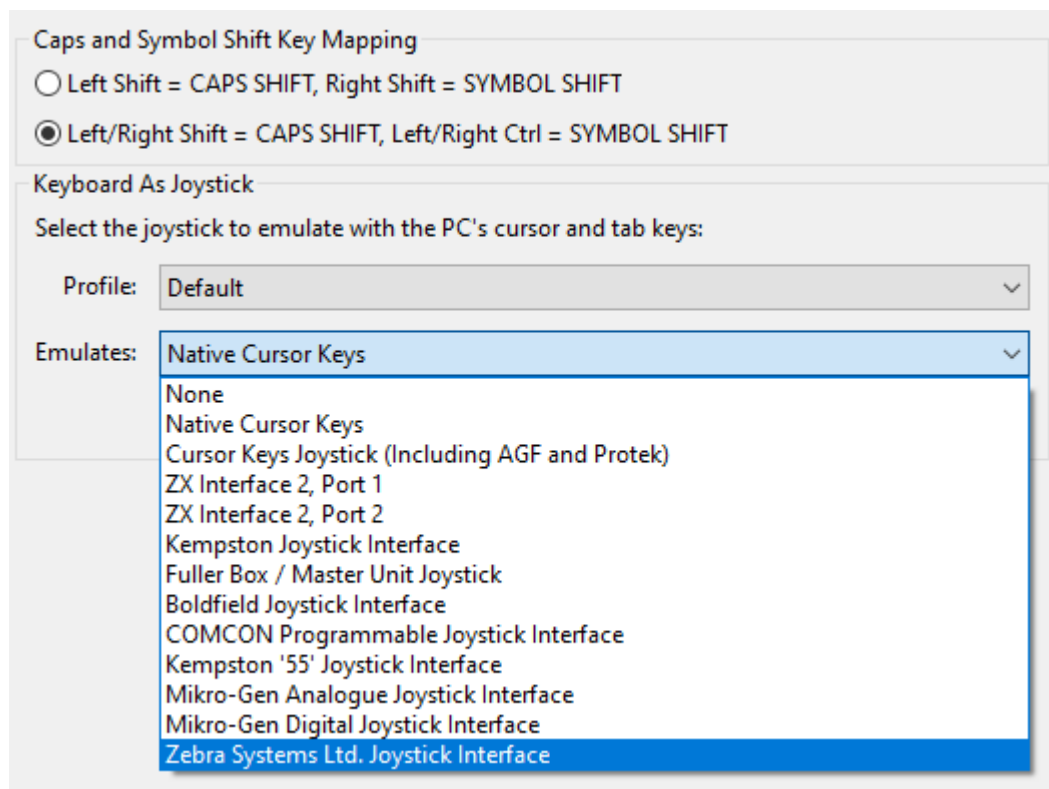
The SHA-1 value shown is the hash used by Inkspector to identify the file being customised. The filename is not used to identify files to be customised this way. "<chuckie.rzx>" is just for reference, but Inkspector identifies files to be customised their hash. i.e. the contents of the file. So if you were to then rename chuckie.rzx to chatty\_egg.rzx and then re-load it, Inkspector would still recognise it as the file you wanted to customise because the contents are the same even though the filename's different.

Displayed underneath the SHA-1 sum is the model of the machine in the snapshot being customised.

The Controller Profile drop-down allows a particular keyboard profile (see [Keyboard](#)) to be associated with the selected file too.

If you decide you no longer want to customise a particular file this way, you can select the file from the drop down (i.e. one with "<filename>" rather than a machine model name) and then click the Forget button.

## Keyboard



The screenshot shows a configuration window with two main sections. The first section, 'Caps and Symbol Shift Key Mapping', contains two radio button options: 'Left Shift = CAPS SHIFT, Right Shift = SYMBOL SHIFT' (unselected) and 'Left/Right Shift = CAPS SHIFT, Left/Right Ctrl = SYMBOL SHIFT' (selected). The second section, 'Keyboard As Joystick', has a heading 'Select the joystick to emulate with the PC's cursor and tab keys:'. Below this is a 'Profile:' dropdown menu set to 'Default'. Underneath is an 'Emulates:' dropdown menu which is open, showing a list of options: 'Native Cursor Keys' (highlighted), 'None', 'Native Cursor Keys', 'Cursor Keys Joystick (Including AGF and Protek)', 'ZX Interface 2, Port 1', 'ZX Interface 2, Port 2', 'Kempston Joystick Interface', 'Fuller Box / Master Unit Joystick', 'Boldfield Joystick Interface', 'COMCON Programmable Joystick Interface', 'Kempston '55' Joystick Interface', 'Mikro-Gen Analogue Joystick Interface', 'Mikro-Gen Digital Joystick Interface', and 'Zebra Systems Ltd. Joystick Interface'.

This page allows the mapping of the physical keyboard to the emulated machine to be customised. The first two options control which PC keys map to the emulated machine's CAPS and SYMBOL shift keys.

Under the Keyboard As Joystick section, you can select how the PC's keyboard is mapped to the emulated machine. Native Cursor Keys refers to the cursor keys used to edit commands by the system's native command editor.

Because I envisaged people wanting to have multiple configurations, for different games, etc., different profiles can be created, each with their own keyboard and joystick settings which can be selected quickly, and also automatically be selected by selecting one on the [Peripherals](#) options page.

### COMCON Programmable Joystick Interface

Selecting the COMCON Programmable Joystick Interface allows you to map the PC's cursor and TAB keys to any of the native machine's keys. This can be useful when playing a game that only has keyboard support, and has a bizarrely selected choice of control keys. This optional essentially allows you to redefine the PC's keys to any you wish. When this option is selected, the Config COMCON button is enabled, allowing you to define the PC's keys as following:

COMCON Configuration

Select how the controller's direction and buttons map to the emulated machine's keyboard. Press CTRL to select Caps Shift and SHIFT for Symbol Shift.

Controller: Keyboard  
Profile: Default  
Emulated Joystick: COMCON Interface

Keyboard	Assigned To
Up	Q
Down	A
Left	O
Right	P
Button 1	M

Reset To Defaults

OK

## Joysticks

Player 1

Select the joysticks to emulate using physical controllers

Controller: Microsoft SideWinder Plug \_Play Game Pad  
Profile: Default  
Emulates: Kempston Joystick Interface

Manage Profiles
Configure Controller

Player 2

Controller: USB gamepad  
Profile: Default Profile  
Emulates: Native Cursor Keys

Manage Profiles
Configure Controller

Inkspector supports the use of up to two game controllers. Each controller can be configured to emulate the same devices as available with [Keyboard](#).

## Display

User Interface

Theme (experimental) Default ▾

Changing this option will require Inkspector be restarted to take effect

Machine Display

Render the machine's display using Direct2D ▾

GDI+ is not recommended for general use due to its lack of hardware acceleration.

On Windows 10 or later, it is possible to select a dark or light theme for the Inkspector GUI. The default of...um...‘Default’ does not select a theme and runs in the default theme for Windows. This is a new feature courtesy of the great [wxWidgets](#) team. It is considered “experimental” as the code to support dark themes in wxWidgets is new, and does have a few known issues at the moment, but it appears to be working fine with Inkspector’s use of wxWidgets.

The second option determines the method used to present, or render, the emulated machine’s display. I recommend always using Direct2D. GDI is provided is a fall-back in case there are issues using Direct2D. GDI+ is provided as an option in case it works better than the others on your machine, but usually it’s noticeably slower than the other two methods and is not recommended for that reason.

## Audio

Volume

☒ Hear beeper 0 100

☒ Hear AY-8912 chips (128k Spectrum, Fullerbox, etc.) 0 100

☒ Hear tape playback and recording noise 0 20 100

☒ Hear speech synthesis 0 100

☒ Hear samples (e.g. SpecDrum) 0 100

All None Default Volumes

Options

☒ Use beeper filtering

☒ Show speech synthesis subtitles in the message window

This page allows the volume of different sources of audio to be controlled individually.

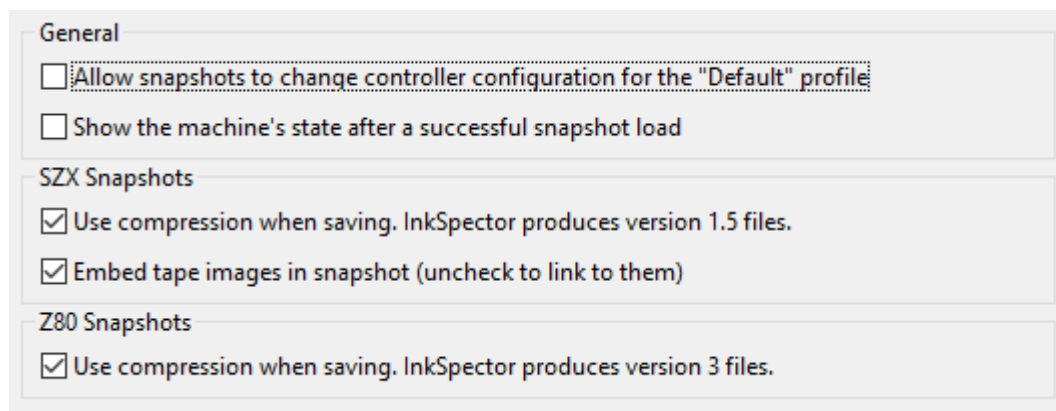
In addition “Use beeper filtering” generates a better quality of beeper sound, particularly so with multi-channel beeper music, and is recommended that this is left checked.

“Show speech synthesis subtitles...” causes each phoneme played by a speech device (e.g. a Currah Microspeech) to be displayed in the message window. For example, when playing Chuckie Egg with a Fuller Orator box attached, might show something similar to the following when a level starts:

(GG2)(OW)[PA1]  
(NN2)(AW)[PA1]

i.e. “Go now!”. Pauses inserted into the speech are shown in square brackets.

## Snapshots



“Allow snapshots to change controller configuration...” allows snapshots to change the “Default” controller profile settings. e.g. with this setting enabled, loading a snapshot that has a Kempston Joystick will set the “Default” profile to use a Kempston Joystick.

“Show the machine’s state...” displays a full dump of a machine’s state after the snapshot has been loaded successfully.

## SZX Snapshots

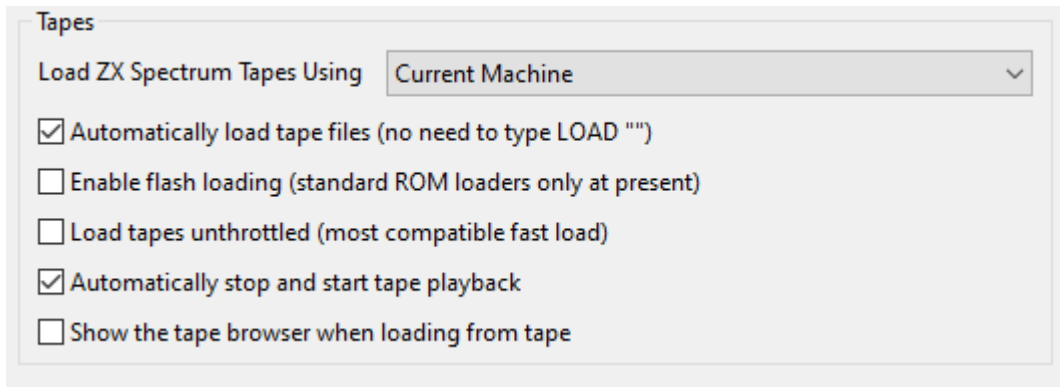
“Use compression when saving...” causes Inkspector to produce compressed .szx snapshot files. There’s no real reason to have this unchecked unless you’re experiencing weird problems.

“Embed tape images in snapshot...” causes any tape image currently loaded into the [Cassette Player](#) to be embedded in the .szx snapshot, rather than storing the tape image file as a link within it.

## Z80 Snapshots

“Use compression when saving...” causes Inkspector to produce compressed .z80 snapshot files. As with the .szx compression option, there’s no real reason to have this unchecked unless you’re experiencing problems with saving .z80 snapshots.

# Tapes



Tapes

Load ZX Spectrum Tapes Using Current Machine

- ☒ Automatically load tape files (no need to type LOAD "")
- ☐ Enable flash loading (standard ROM loaders only at present)
- ☐ Load tapes unthrottled (most compatible fast load)
- ☒ Automatically stop and start tape playback
- ☐ Show the tape browser when loading from tape

The first option controls which type of machine is used when loading a tape image file.

“Automatically load tape files” causes the machine to be rebooted (using the type of machine selected above) and the appropriate LOAD command automatically typed in on the machine has booted up.

“Enable flash loading” enables instant loading of programs that use the standard ROM loaders

“Load tapes unthrottled” temporarily unthrottles the emulated machine, running it as fast as possible, until the tape has finished loading, when the machine’s speed is throttled again. This is the most compatible way to load a tape at maximum speed at present.

“Automatically stop and start tape playback” automatically pauses and unpauses tape playback as the tape loader requires it.

“Show the tape browser...” automatically displays the [Tape Browser](#) when loading from tape is detected.

# Recordings

**Animated GIFs**  
☒ Include the Spectrum's border in the output  
☒ Use transparency optimisation  
☒ Use sub-window optimisation  
Produce GIFs with a frame rate of  frames per second.

**RZX Recordings**  
☒ Use compression when producing RZX files  
☒ Use last-frame optimisation when producing RZX files  
☐ Rebuild snapshots when importing RZX files  
☒ Automatically create rollback points, every  minutes

**AV Recordings**  
AAC Average Bytes / Sec  bytes/sec  
Video Bit Rate  bits per second Video Profile   
☐ Destination Width  ☐ Destination Height   
☐ Include the Spectrum's border in the output

## Animated GIFs

This group of options controls how animated GIFs are produced, starting with whether the Spectrum's border is included in the GIF (obviously this option has no effect when recording a machine that doesn't have a border).

"Use transparency optimisation" and "Use sub-window optimisation", when enabled (which is the recommended setting for both) help produce the smallest animated GIF files. These options are provided here only to aid diagnostics should the resulting .gif file not play back correctly in some software.

The final GIF option is to specify the frame rate of the resulting .gif file. Obviously the more frames per second you have, the larger the resulting file.

## RZX Recordings

"Use compression when producing RZX files" produces smaller .rzx files by compressing the data that represents the player's joystick and keyboard inputs.

The RZX format allows for an optimisation when the keyboard and joystick input data is the same as the previous frame (the last frame optimisation). Check this option to have Inkspector use this optimisation creating RZX recordings.

Checking "Rebuild snapshots when importing RZX files" has Inkspector rebuild each snapshot that is imported from an RZX file. If the original .rzx file contains embedded .z80 snapshots, having Inkspector rebuild them (as .szx files) can usually reduce the size of the resulting recording. If the

recording being imported already uses szx files, it's possible the resulting recording could be slightly larger.

When making an RZX recording, it's useful to have rollback points in case you make an error a long way into a game. Checking the "Automatically create rollback points" and selecting how often this should occur in minutes will take care of creating these points for you. Regardless of this setting, you can manually create rollback points as often and as many times as you like.

## AV Recording

This group of options controls how mp4 video files are produced, including whether the Spectrum's border is included in the mp4 file (just as with Animated GIFs, this option has no effect when recording a machine that doesn't have a border).

The default audio and video recording rates are specified here, along with the video profile. The "main" profile is usually sufficient for most uses, but "baseline" and "high" are also provided in case of specific video requirements.

Finally the destination width and height of the video can be set. This provides a mechanism to scale the video output. For example, if recording a Spectrum without a border (i.e. width of 256, height of 192), overriding the width to 512 and height to 384 will create a video file scaled by 2.0.

## ROM Management (including listing and symbol files)

The screenshot shows a settings window titled "ROM Management (including listing and symbol files)". It is divided into two main sections: "ROM Symbol and Listings" and "Custom ROMs".

**ROM Symbol and Listings:**

- ☒ Load symbols and listing files as configured below
- ☒ Use symbol and listing file cache to improve load performance
- 8 Symbol And Listing Files Are Cached
- Buttons: Clear Cache, View Cache

**Custom ROMs:**

- ROM: ZX Spectrum 16k/48k (dropdown menu)
- Default File: 48.rom
- ☒ Use custom ROM:
- Text field: C:\temp\gosh\gw03.rom
- Button: Browse
- The following options may be used to enable source-level debugging of ROMs
- ☒ Use symbol file:
- Text field: C:\temp\gosh\gw03.sym
- Button: Browse
- ☒ Use listing file:
- Text field: C:\temp\gosh\gw03.lst
- Button: Browse
- Files produced by: Automatically Detect Assembler (dropdown menu)

Custom ROMs for all supported machines and peripherals may be specified on this screen. In the screenshot above, we're specifying the 16 and 48K Spectrums use Geoff Wearmouth's GOSH WONDERFUL ZX Spectrum ROM (gw03.rom) instead of the standard ROM.



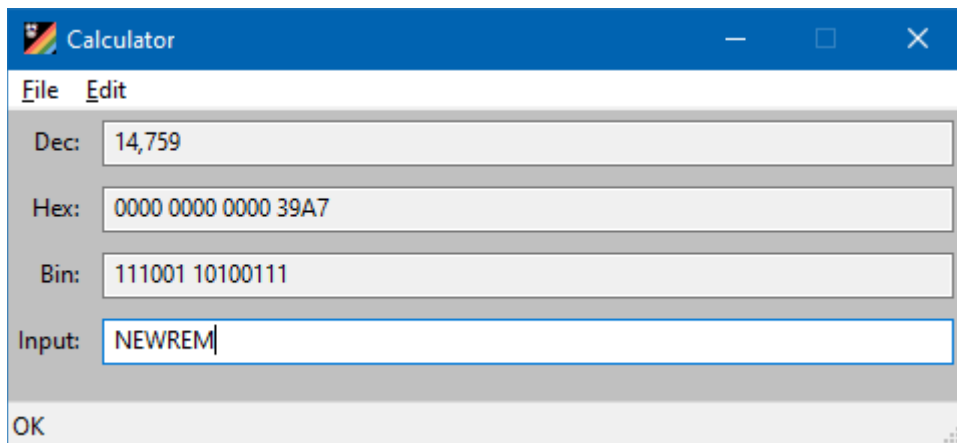
Inkspector is able to load symbol and listing files produced by the TASM, SjAsmPlus and PasmO assemblers for each custom ROM to aid debugging.

Symbol files that are loaded have their contents made accessible anywhere where a numeric value can be entered within Inkspector. For example, using the gw03 source code (I used [https://k1.spdns.de/Vintage/Sinclair/82/Sinclair%20ZX%20Spectrum/ROMs/gw03%20%27gosh%2C%20wonderful%27%20\(Geoff%20Wearmouth\)/gw03%20rom%20source.txt](https://k1.spdns.de/Vintage/Sinclair/82/Sinclair%20ZX%20Spectrum/ROMs/gw03%20%27gosh%2C%20wonderful%27%20(Geoff%20Wearmouth)/gw03%20rom%20source.txt) renamed to gw03.asm), run TASM as follows to produce the .rom, .lst and .sym files from it:

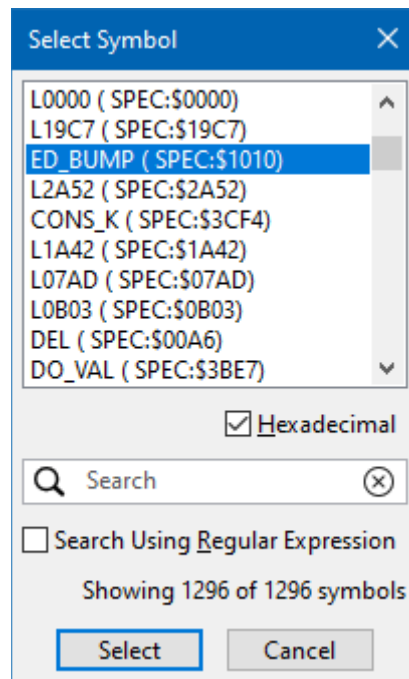
```
tasm -80 -s -b gw03.asm gw03.rom
```

Then use the screen above to select gw03.rom as a custom ROM and gw03.sym and gw03.lst as the symbol and listing files respectively. Make sure the top check box (“Load symbols and listing files...”) is checked otherwise the symbol and listing files will not be used.

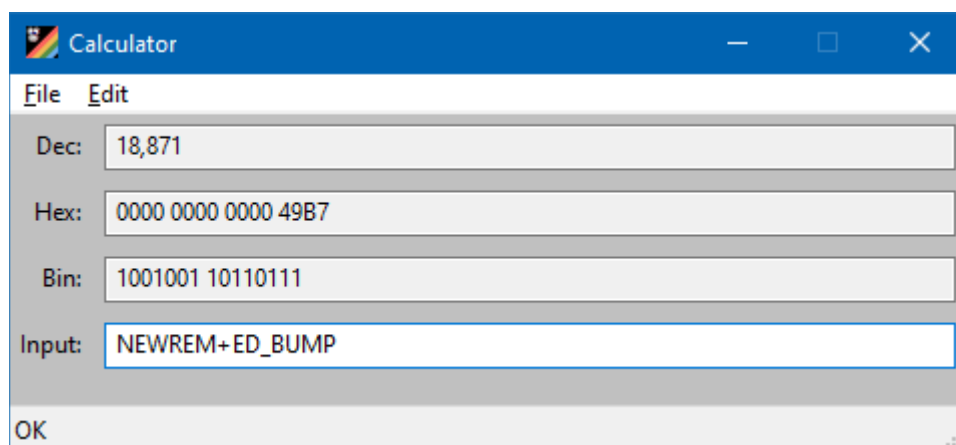
If you then choose View → Calculator and type in NEWREM, it should show the value 14759 as below:



If you then type +, to end up with “NEWREM+” , you should get a syntax error. Fear not. From the Calculator’s Edit menu, click Select Symbol (or just press CTRL-S). This is a common dialog window that is available from many places in Inkspector (often from a “Symbol” button), and it gives you access to all the symbols that Inkspector currently has loaded and allows one to be selected.



Select a label of your choice then press the “Select” button. I’m going to look for the snappily named ED\_BUMP symbol and select that (you can also type in the Search area and the display will show all matching results as you type). After pressing “Select” the Calculator then shows:



Listing files allow the debugger to show source code extracted from listing files, to provide more information than showing just the raw disassembly would do. Similarly, supplying a symbol file allows the debugger (and anywhere else in Inkspector where a numeric value may be entered) to access them.

For example, looking at the first few instructions of the gw03 ROM (which is the same as the original Spectrum ROM for this section) in the debugger usually looks something like this:

\$0000	f3	di	
\$0001	af	xor	a
\$0002	11 ff ff	ld	de,\$ffff
\$0005	c3 cb 11	jp	\$11cb
\$0008	2a 5d 5c	ld	hl,(\$5c5d)
\$000B	22 5f 5c	ld	(\$5c5f),hl
\$000E	18 43	jr	\$0053

When a symbol file has been loaded, they are incorporated into the display (the L0000 and L0008 label headers and the using of the label L11CB rather than address \$11CB as above) as follows:

```
L0000:
| $0000  f3      di
  $0001  af      xor    a
  $0002  11 ff ff ld     de,$ffff
  $0005  c3 cb 11 jp     L11CB

L0008:
  $0008  2a 5d 5c ld     hl,($5c5d)
  $000B  22 5f 5c ld     ($5c5f),hl
  $000E  18 43   jr     L0053
```

And when the gw03.lst file is also selected as shown above, the debugger display for this section of memory then looks like:

```
L0000:
$0000  f3      DI          ; Disable Interrupts.
$0001  af      XOR    A    ; Signal coming from START.
$0002  11 ff ff LD      DE,$FFFF ; Set pointer to top of possible physical RAM.
$0005  c3 cb 11 JP      L11CB ; Jump forward to common code at START-NEW.

L0008:
$0008  2a 5d 5c LD      HL,($5C5D) ; Fetch the character address from CH_ADD.
$000B  22 5f 5c LD      ($5C5F),HL ; Copy it to the error pointer X_PTR.
$000E  18 43   JR      L0053 ; Forward to continue at ERROR-2.
```

Returning back to the ROM Management options page, checking the “Use symbol and listing file cache...” option instructs Inkspector to cache all listing and symbol files (actually it’s the internally processed version of these files that’s cached to further improve performance) for all machines and peripherals, updating the cache automatically whenever the files change. When unchecked, each listing and symbol file is loaded and processed when Inkspector starts up, which can start to slowdown Inkspector’s start-up times if many listing and symbol files are in use. I’ve yet needed to uncheck this option, but it’s provided pre-emptively as a diagnosis tool.

The “Clear Cache” button clears the cache of all listing and symbol files.

The “View Cache” button displays a list of items in the listing and symbol cache.

# Assembler

Confirmation

☒ Show successful assembly confirmation (errors are always shown)

Errors

☒ For assembler errors with .s and .asm files, run the following program:

C:\wscite\SciTE.exe

Browse

Command Line Options: \$(PATH) -goto:\$(LINE),\$(COL)

Use \$(PATH), \$(LINE) and \$(COL) to specify filename, line number and column

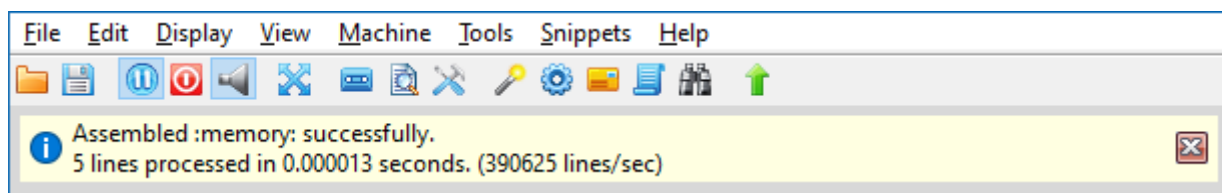
Auxiliary Snapshot Files

These files are created by the assembler and enable its BREAK directive to work when one of its snapshot files is loaded. They have a file extension of .aux.xml

☒ Use auxiliary snapshot files

## Confirmation

Ticking the first checkbox causes a message to be displayed on a successful assembly, whether it was started by loading a .s or .asm file, or launching the [assembler via script](#). For example, after a successful assembly via script with this option enabled:



And as the description says, assembly errors are always shown, regardless of this setting.

## Errors

Checking this box causes the specified program to be run whenever assembling a .s or .asm file fails (but not when invoking the assembler directly from script with [inks.assemble](#)).

Any program may be specified, with user supplied optional command line options. Using any of the following special options will cause them to be substituted for the values taken from the location of the first assembler error:

Option	Expands To
\$(PATH)	The full path of the file where the assembly error occurred
\$(LINE)	The line number within the file where the assembly error occurred
\$(COL)	The column within the line, within the file where the assembly error occurred.

If you use the Browse button to select the SciTE editor (scite.exe), Inkspector will set command line options for you automatically, so that Scite will open the file at the exact place the assembly error occurred, e.g. command line options

```
scite.exe $(PATH) -goto:$(LINE),$(COL)
```

Expands to

```
scite.exe my_prog.s -goto:123,20
```

## Auxiliary Snapshot Files

One of my favourite recent features of Inkspector is its [BREAK](#) assembler directive, that sets debugger breakpoints on successful assembly. When running the resulting assembly in the debugger (e.g. after loading a .s or .asm file) the breakpoints are created and take effect automatically. However, if the assembly creates, for example, a .szx snapshot file, although the BREAK breakpoints will still take effect as the assembly is run in the debugger, loading the resulting snapshot separately will not activate the breakpoints created by any assembler BREAK directives. This is where auxiliary snapshot files come in.

If the option to use auxiliary snapshot files is enabled (it's disabled by default), and BREAK directives are used, each of the generated snapshot files will also have an auxiliary snapshot file generated that is paired with the main snapshot file. The filename of such auxiliary files is <snapshot filename>.aux.xml. For example, if the assembly generates a snapshot called ace\_game.szx, the auxiliary snapshot file will be ace\_game.szx.aux.xml.

Currently, the auxiliary file contains only BREAK breakpoint information, so the breakpoints are seamlessly restored on loading the snapshot file generated by the assembler (remember that the BREAK directive does not inject any special Z80 codes into the resulting assembled code, so that BREAK may be used in 'production' code).

Although these auxiliary snapshot files currently store only BREAK breakpoint information, they have the potential to add general snapshot enhancements, such as adding information for peripherals that aren't supported by the snapshot natively, etc.

# Debugger

Symbols and Source Code

☒ Expose the machine's system variables as symbols

When a snapshot or tape image is loaded:

☐ Attempt to extract source code from a .lst listing file (except Pasm0)

☐ Attempt to load symbols from a .sym symbol file

Symbol files produced by: Automatically Detect Assembler

Disassembly

☒ Display values in hexadecimal (affects all windows)

☒ Underline hyperlinks

☐ Show the disassembly in uppercase

☒ Show blank lines after a branch

☒ Show single step times against instructions

☒ Show at end of line

Breakpoints

☒ Automatically save and restore breakpoints between sessions

☒ Show a message on the emulator window when breakpoints are hit

Display

☒ Show all writes to the display immediately

## Symbols and Source Code

Checking the first box, allows the [system variables](#) for the current machine to be used anywhere a symbol or expression may be entered. For example, with this option checked and running a 48K Spectrum, adding 'FRAMES' to the [Watch](#) window evaluates to \$5C78:

Watch		
File Edit View		
Watch	Value	
FRAMES	\$5C78	

Just as the Inkspector debugger can use symbol and listing files for system and peripheral ROMs to [improve the debugger's Z80 code display for custom ROMs](#), it can also automatically attempt to load such files for a snapshot that's loaded, by looking a files with .sym and .lst extensions sitting beside the snapshot file loaded.

For example, with 'Attempt to extract source code from a .listing file' checked, loading the snapshot file c:\temp\mayhem.sna would attempt to load c:\temp\mayhem.lst and extract the source code from it, just as it can for system and peripheral ROMs. Similarly, checking 'Attempt to load symbols...' would attempt to load c:\temp\mayhem.sym and extract symbol information from such a file.

And just as with the ROM Management screen, you can select from the drop down list which assembler produced the .sym files, in case the default automatic assembler detection needs a steer.

## Disassembly

‘Display values in hexadecimal’ turns on hexadecimal display of numbers throughout Inkspector and takes effect in all windows as soon as OK is pressed. Uncheck this box to display numbers in decimal.

‘Underline hyperlinks’ displays all hyperlinks underlined so they’re easily identifiable.

Check ‘Show the disassembly in uppercase’ if you prefer to see “LDIR” rather than “ldir” for example

‘Show blank lines after a branch’ displays a blank line after a Z80 instruction that jumps or returns, which makes the code easier to read. For example, with blank lines:

\$10C5	fe 20	cp	<a href="#">\$20</a>
\$10C7	30 52	jr	nc, <a href="#">\$111b</a>
\$10C9	fe 10	cp	<a href="#">\$10</a>
\$10CB	30 2d	jr	nc, <a href="#">\$10fa</a>
\$10CD	fe 06	cp	<a href="#">\$06</a>
\$10CF	30 0a	jr	nc, <a href="#">\$10db</a>

and without:

\$10C5	fe 20	cp	<a href="#">\$20</a>
\$10C7	30 52	jr	nc, <a href="#">\$111b</a>
\$10C9	fe 10	cp	<a href="#">\$10</a>
\$10CB	30 2d	jr	nc, <a href="#">\$10fa</a>
\$10CD	fe 06	cp	<a href="#">\$06</a>
\$10CF	30 0a	jr	nc, <a href="#">\$10db</a>

‘Show single step times..’ shows in red the number of t-states taken to execute the current instruction. The time taken remains displayed on the line until the debugger’s View → Clear Single Step Times menu item is selected.

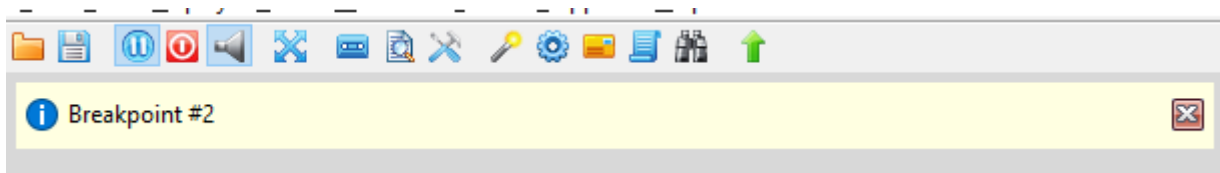
\$0038	f5	push	af (11)
\$0039	e5	push	hl (11)
\$003A	2a 78 5c	ld	hl,( <a href="#">FRAMES</a> ) (16)
\$003D	23	inc	hl
\$003E	22 78 5c	ld	( <a href="#">FRAMES</a> ),hl

Note that the times taken may exceed the basic Z80 timings for an instruction, as any delay introduced by the underlying system (e.g. the Spectrum’s contended memory) is included in the t-state time shown.

## Breakpoints

‘Automatically save and restore breakpoints between sessions’ attempts to save all current breakpoints for the current machine when Inkspector is closed down.

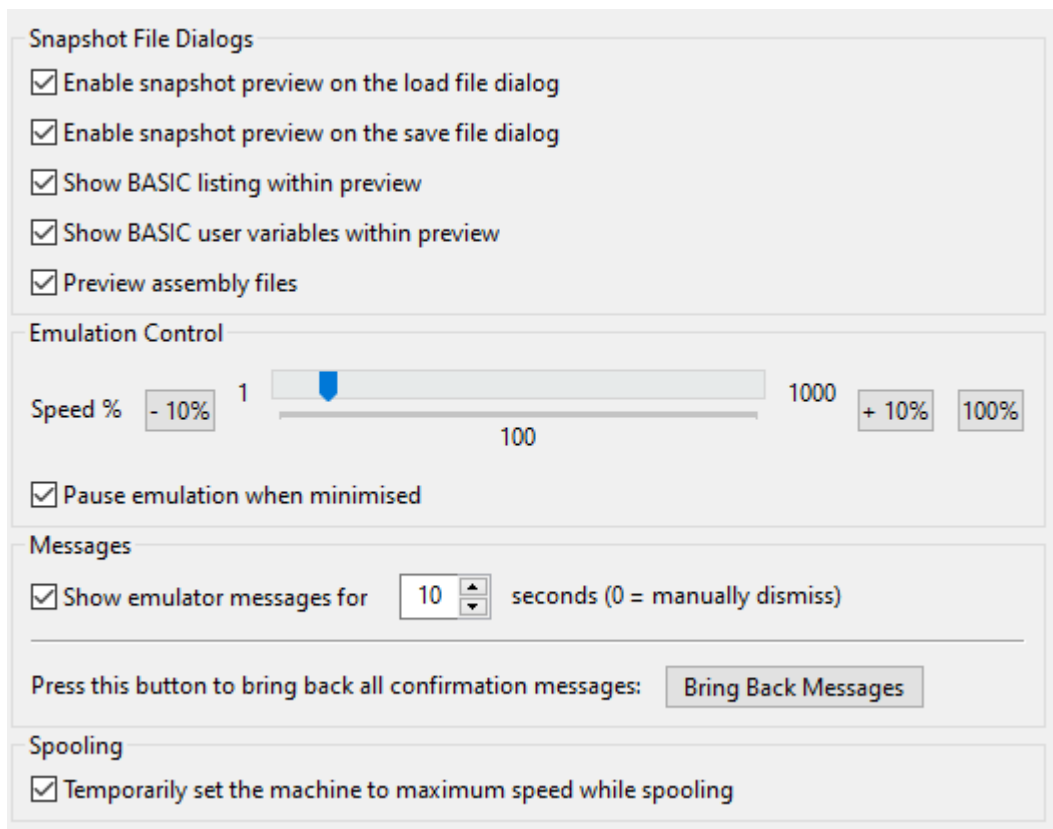
‘Show a message on the emulator window when breakpoints are hit’ confirms the breakpoint hit on the main Inkspector display, e.g.



## Display

Checking 'Show all writes to the display immediately' will cause all memory writes to the machine's display area to be reflected on screen immediately when single-stepping in the debugger. When this option is unchecked, writes to the display that are behind the beam will not be seen until the next frame, as per a real machine.

## General



### Snapshot File Dialogs

This set of options controls whether the live snapshot previews appear on the snapshot load and save file selection dialog windows. In addition you can select whether BASIC listings and variables present in the snapshot are shown.

By default, assembly files (.s and .asm) are not assembled in the preview, but you change this by checking the 'Preview assembly files' box.

### Emulation Control

Here you can change the speed of the emulated machine from 1% to 1000% of the real machine's speed.



### **InkTip**

If you want to make Inkspector emulate the machine as fast as possible, it's easier to uncheck Machine → Throttle Emulation Speed (shortcut key F8).

'Pause emulation when minimised' will pause the emulated machine when the Inkspector GUI is minimised, unpausing automatically when it's restored.

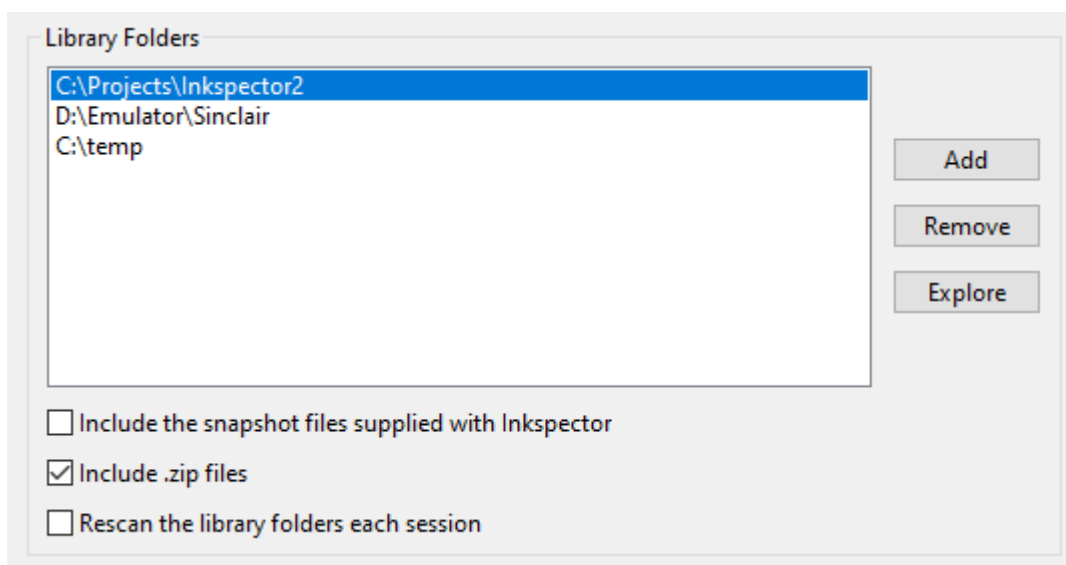
## **Messages**

Normally, non-invasive messages for the user (for example, the completion of playback of an .rzx recording) are shown at the top of the machine's screen for 10 seconds, or until the message's close box is clicked. You can uncheck the 'Show emulator messages' box if you do not want to see these messages, but you might miss something useful! What's more useful is that you can change how long each message is displayed for before it dismisses itself (or use 0 to specify the message will remain on screen until dismissed by the user by clicking on its close box).

## **Spooling**

Checking this option will temporarily set the emulation speed to 'unthrottled' (i.e. run at maximum speed) while a machine is spooling, automatically returning to 'throttled' once the spooling operation has finished, which can be very useful when spooling long text files.

## **Library**



The Library is a collection of folders that can then be manually searched through as a single collection, using [Search Library](#) or querying [via script](#).

To add a folder to the library, click on the Add button. Click on Remove to remove one, or Explore to open an Explorer window for the selected folder.

Inkspector comes supplied with a few snapshots to get started with. If you want these to be included in your library, check the first box.

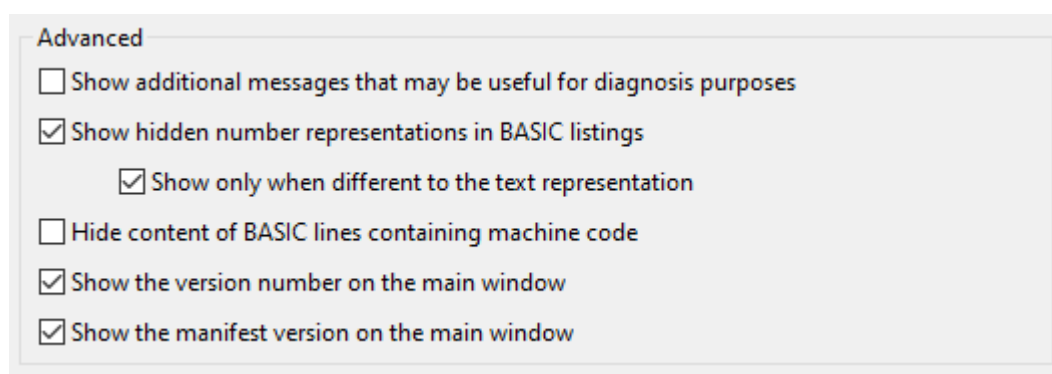
Since Inkspector can load supported files directly from within .zip files, you can have the library include any .zip files that are found within the library folders by checking 'Include .zip files'. Note

that all .zip files regardless of their content will be included in the library, so if you have .zip files within the specified folders that don't include any files supported by Inkspector, they will still be included in the library.

'Rescan the library folders each session' causes the entire library to be rebuilt by rescanning all specified folders the first time the library is accessed for a session of Inkspector. While this does keep the library completely up-to-date with any changes within the library folders, it can take a long time if you have a large library collection, so by default this option is unchecked. In which case you can manually re-scan the library as and when required from the [Search Library](#) window.

If any of the folder libraries become inaccessible (either deleted or renamed, or moved) their paths will be shown in red in the Library Folders list.

## Advanced



Advanced

- ☐ Show additional messages that may be useful for diagnosis purposes
- ☒ Show hidden number representations in BASIC listings
  - ☒ Show only when different to the text representation
- ☐ Hide content of BASIC lines containing machine code
- ☒ Show the version number on the main window
- ☒ Show the manifest version on the main window

Checking 'Show additional messages that may be useful...' causes additional messages to be displayed on the message window. If you're investigating some problem with Inkspector or you're having issues loading a snapshot or other supported file, checking this option may be able to shed additional light on it. This is equivalent to using the `--verbose` option with the CLI.

Whenever Inkspector displays a BASIC program, it can include the hidden number representations (not normally visible to the user) that are stored alongside the visual representations. Additionally it can display the hidden versions only when they differ from the visual (text) representation, which is usually used as an [anti-hacker measure](#).

'Hide content of BASIC lines containing machine code' hides lines that *appear* to contain only embedded machine code rather than BASIC commands. The algorithm used to determine this can occasionally produce false positives, which is why it is turned off by default, but if you're examining a program that does contain lots of embedded machine code, this option can be useful.

The final two options on this page control whether the Inkspector version number (e.g. 2.0.7) and / or the manifest version (The [Fossil](#) commit hash of the check-in used to create the Inkspector executable, e.g. [810cc181ee]) are shown on the main window's title bar.

## Logging

The screenshot shows the 'Logging' section of the Inkspector application. It contains two main panels. The top panel, titled 'Logging', has two checkboxes: 'Send log messages to debug output' and 'Record all messages to database'. To the right of these checkboxes are three buttons: 'Info.', 'Clear', and 'Export'. The bottom panel, titled 'Diagnostic Logging', contains a warning message: 'Enabling these may slow down Inkspector as they can produce lots of information.' Below this message is a grid of ten dropdown menus, each set to 'None'. The dropdowns are arranged in two columns: Hardware access, Assembler, Tape loader, Floppy controller, AY and Speech, Events, Debugger, ZX Interface 1, Game Controllers, and Snapshot Processing. At the bottom right of the 'Diagnostic Logging' panel is an 'All None' button.

Logging	
<input type="checkbox"/> Send log messages to debug output	
<input type="checkbox"/> Record all messages to database	
	Info. Clear Export

Diagnostic Logging	
Enabling these may slow down Inkspector as they can produce lots of information.	
Hardware access	None
Assembler	None
Tape loader	None
Floppy controller	None
AY and Speech	None
Events	None
Debugger	None
ZX Interface 1	None
Game Controllers	None
Snapshot Processing	None
All None	

## Logging

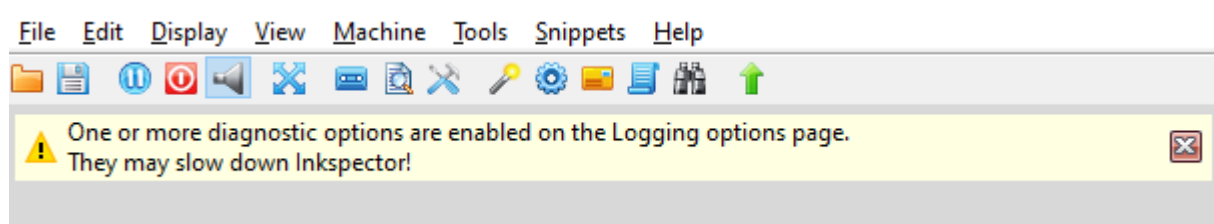
Checking 'Send log message to debug output' will send each message sent to the [message window](#) to the Windows debugger too, using the [OutputDebugStringW\(\)](#) function, where it can be seen within the Visual Studio debugger or outside of Visual Studio by using a separate tool such as the Sysinternals [DebugView](#) tool.

Checking 'Record all messages to database' will also copy each message sent to the message window to the Inkspector database. This is occasionally useful for me when doing testing, but isn't expected to be particularly useful to others.

Clicking the "Info." button will display how many messages have been stored in the database. "Clear" clears any stored within the database (or greyed out if there are none). "Export" allows all messages stored in the database to be exported to an ASCII text file.

## Diagnostic Logging

This section allows diagnostic messages to be enabled for various areas of Inkspector. Enabling even just one may produce a lot of additional information, and can in some cases noticeably slow down Inkspector due to the sheer volume of additional messages produced. This is why a reminder is shown when Inkspector is started when one of these options is enabled (i.e. set to something other than 'None').

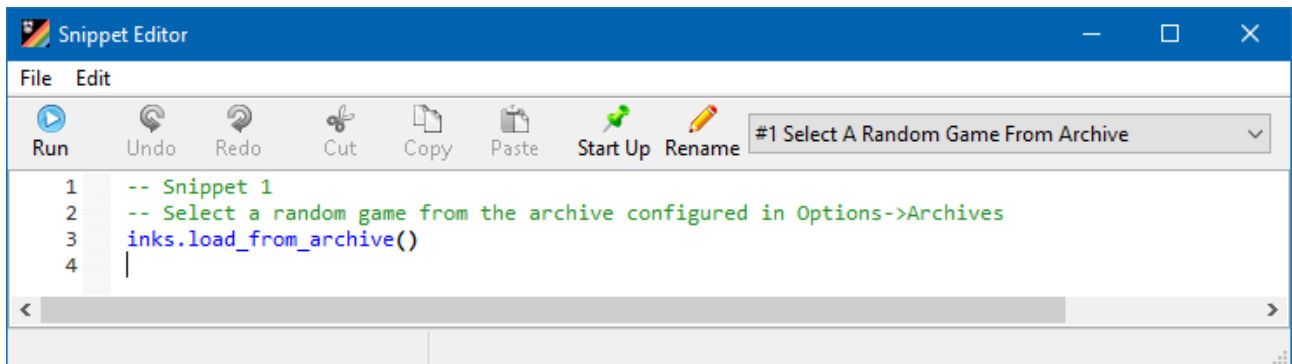


# Snippets Menu

The Snippets menu provides access to the Snippet editor (also available from the main window toolbar by clicking on the scroll or pressing Alt-E) and the 10 snippets, whose keyboard shortcuts are Alt-1 through to Alt-0.

## Snippet Editor

The Snippet editor is where the ten snippets are edited. They are selected by clicking on the drop-down list on the toolbar. Any changes made to the current snippet are automatically saved when doing this.



Any one of the ten snippets may be set as the start-up snippet that is run automatically when Inkspector starts. You can set it from within this editor by clicking on the Start Up toolbar button, which will remain depressed to indicate it's set. The start-up snippet may also be set from the [Options page](#). Note that if Inkspector is started with a filename specified on the command line, the start-up snippet is not run. This is to avoid a situation where the file specified is loaded but then subsequently discarded by the snippet that might create a new machine or load another file.

You can give the snippets meaningful names (which are shown on the main window's Snippets menu too) by clicking on the Rename toolbar button.

If you attempt to close the Snippet Editor while there's an error in one of the snippets, you will be presented with the first error and taken back to the offending snippet to correct. If you really need to exit the editor with an error remaining, hold down the CTRL key while closing the editor which bypasses the code check.

If you wish to reset the snippets back to their factory defaults, click on the Edit menu and select Reset Snippets. After confirming the operation, the current snippets will be replaced with the factory defaults.

### InkTip

The name of the Inkspector functions in the inks table are highlighted in blue (as shown above with `inks.load_from_library()`), to provide immediate feedback as to whether the function name is recognised. The names of constants within the inks table are highlighted in italic blue.

### InkTip

The default code [snippet](#) #5 (therefore keyboard shortcut Alt-5) toggles the global setting between

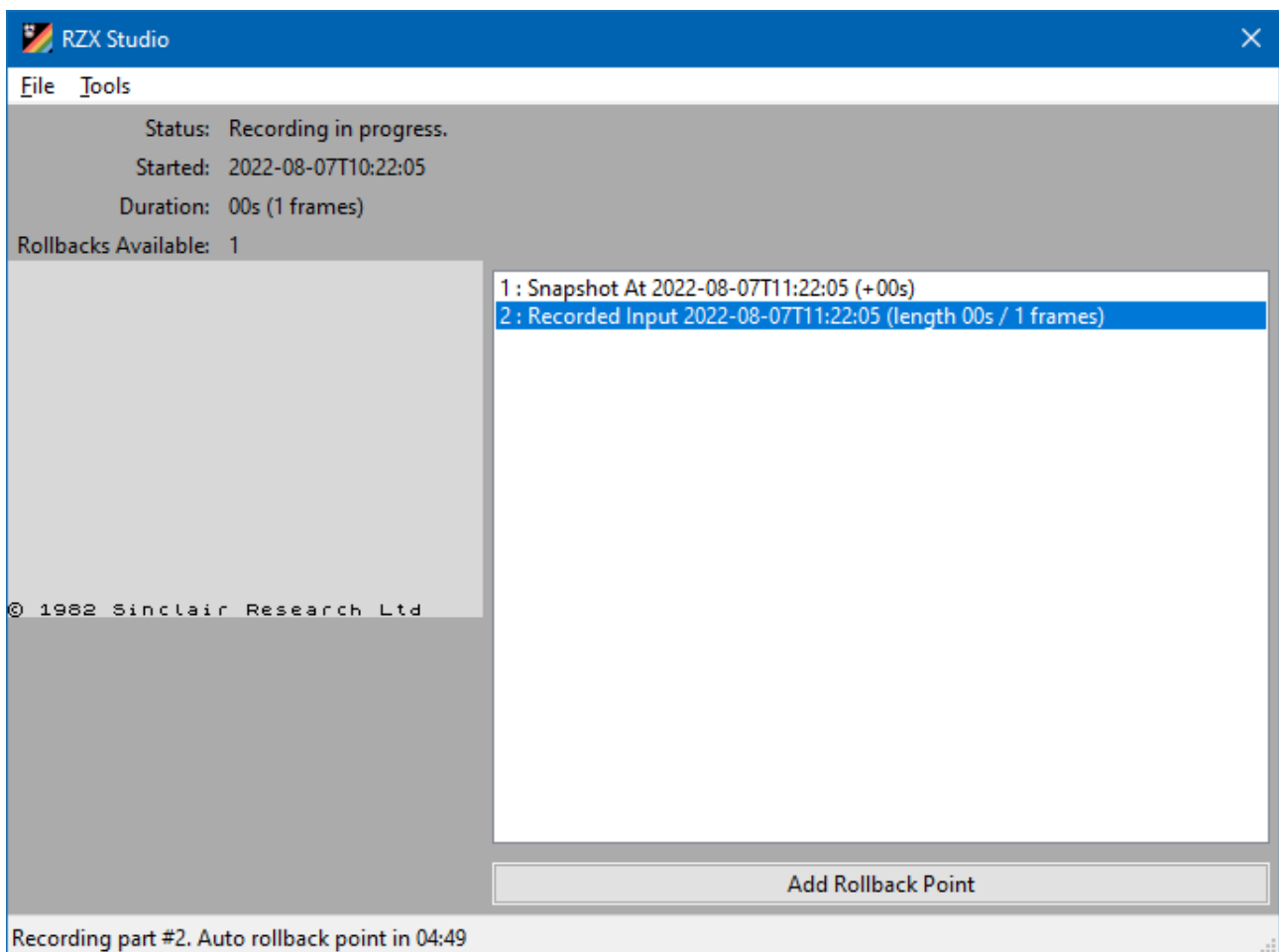
hexadecimal and decimal.

## RZX Studio

The ostentatiously titled RZX Studio is where RZX recordings are created or existing ones imported, either to record on to the end of them or to export one of their embedded snapshots.

### Creating A New RZX Recording

To create a new RZX recording, make sure you're running one of the ZX Spectrum models (RZX recordings cannot be created for the ZX80, ZX81 or Jupiter ACE) and select File → Start New Recording. If an existing recording is present you will be asked whether you wish to delete it first. When the recording begins, you will see a thumbnail of the Spectrum display



Top to bottom, the information shown is: the current status of the RZX recorder, the time and date when the recording was started, the duration of the recording as of the current rollback point (so doesn't necessarily include any in-progress part of the recording) and the number of rollback points available.

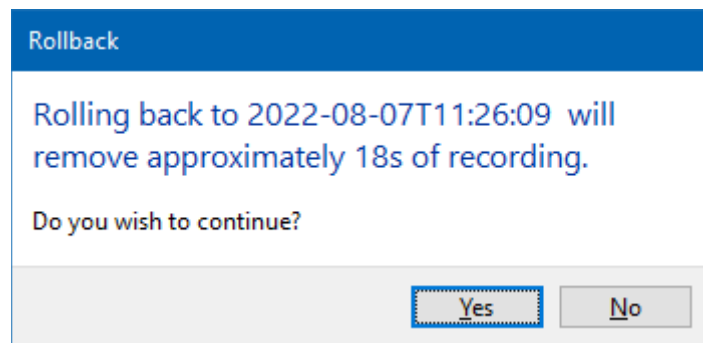
The list on the right hand side shows all the component parts of the recording, which always starts with a snapshot to set the starting state of the machine for the recording, followed by the actual input by the user which is stored separately as 'Recorded Input'. Each rollback point added adds

another snapshot part. The thumbnail image on the left represents the first snapshot part above the currently selected part of the recording.

As well as the RZX Studio window status bar showing “Recording” and a countdown timer showing how long before the next automatic rollback point is added, the main window also shows “Rec.” and how long the recording currently is.

Before taking a leap on a particularly a tricky platform game, you may wish to add a rollback point, which as its name suggests, allows you to rollback the recording to a previous rollback point (but not the very first point of the recording – that would essentially be deleting the recording) should you misjudge Willy’s jump and end up in the Entrance to Hades. By default, rollback points are created every 5 minutes as a safety net, but you can add as many manual ones as you desire by pressing the “Add Rollback Point” button.

To rollback, first select the rollback point on the recording list then select ‘Rollback’ from the File menu. You will be informed how much of the recording will be deleted if you wish to continue with the rollback, e.g.



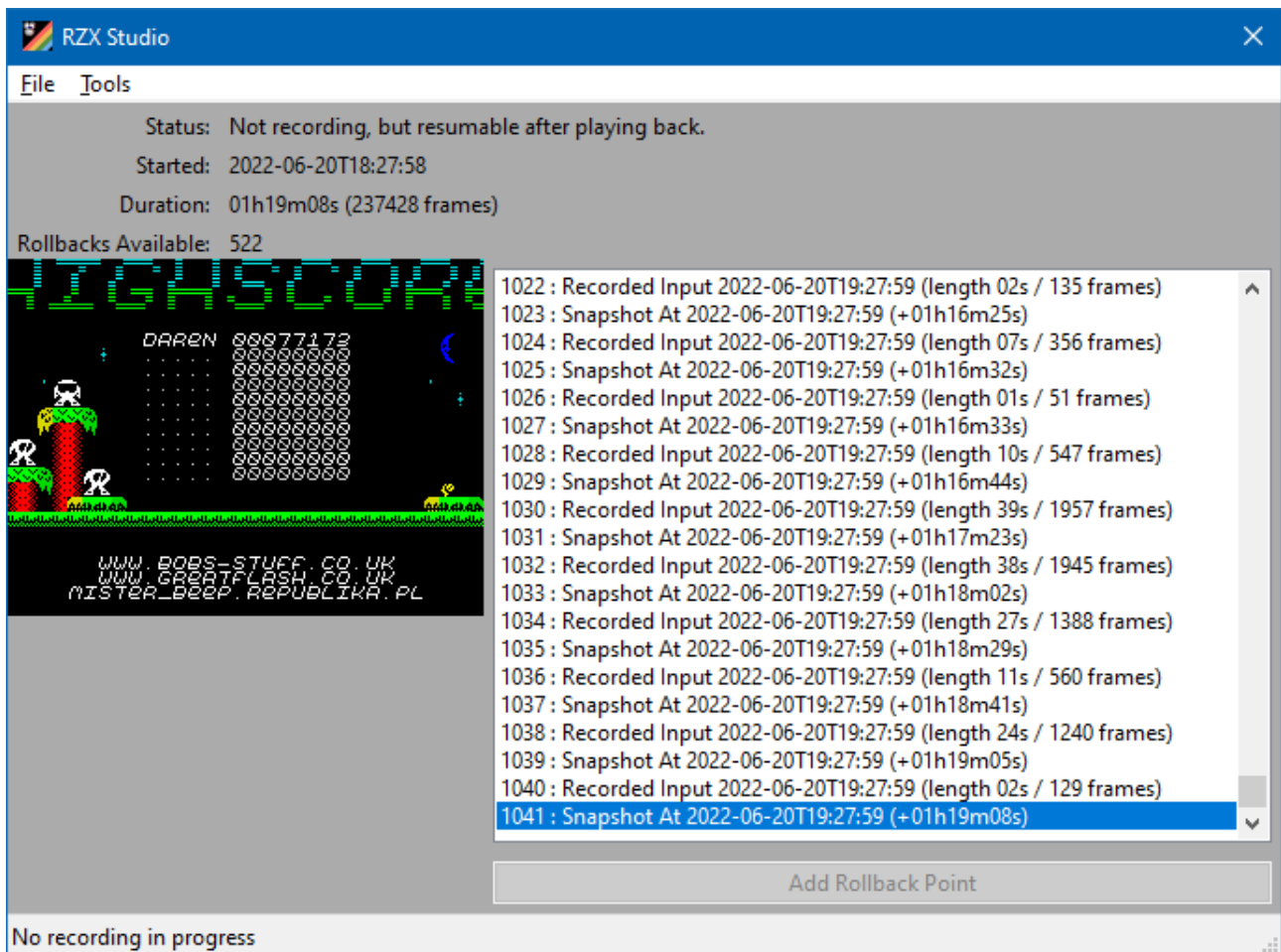
Selecting ‘Yes’ will rewind the recording to the rollback point and resume recording immediately.

From the File menu, you can also stop the recording (with a view to resuming or creating a .rzx file from the recording made so far), resume recording, resume recording from a snapshot which will add the selected snapshot to the recording and continue from that point, rollback, save the recording as a .rzx file, revert the recording to how it was when Inkspector was started, or delete it completely.

Note that in-progress recordings are saved automatically when you close Inkspector (as to avoid any #!@\*&% outbursts when you realise you didn’t save it explicitly before closing), so if you really don’t want to keep a recording you will need to delete it by using the File menu’s ‘Delete Recording’ item.

## Importing An Existing RZX Recording

From RZX Studio’s Tools menu, click on Import RZX File and select the .rzx file you wish to import. For the screenshot below, I used horacewoods\_hard.rzx as it contains a whopping 522 embedded snapshots, so it’s a good one to test with. Once the file has imported, you will be presented with something similar to this:



If you click on one of the “Snapshot At...” lines, you can save it out as a separate snapshot file using Tools → Save As Snapshot.

If you wish to record on to the end of this recording, select File → Resume Recording. And when you’re done you can save out the resulting .rzx file just as though you had created the recording from [scratch](#).

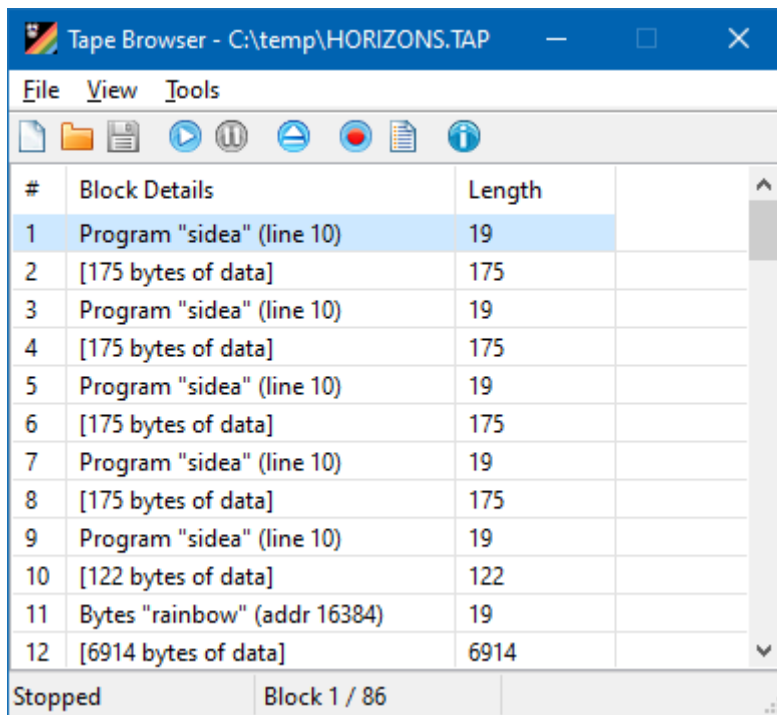
## Tape Browser

The tape browser shows the contents of any tape currently loaded. It’s also where tapes are created. In addition it allows you to view any BASIC programs stored in one or more of the tape’s blocks.

## Loading A Tape

Loading a tape into Inkspector can be done several ways: by passing the name of the tape image file on the GUI’s command line, by dragging a tape image file on to the main window, by using Load Supported File or Recent Tapes menus on the main window, or, on the Tape Browser window, the File and Recent Tapes (which is a copy of the Recent Tapes menu on the main window for convenience) menus or Open Tape File on the toolbar.

Once a tape image file has been loaded, the window will display the list of tape blocks loaded from it. Loading an image of the Horizons tape shows the following:



The screenshot shows the 'Tape Browser' application window with the file 'C:\temp\HORIZONS.TAP' open. The window has a menu bar (File, View, Tools) and a toolbar with icons for file operations and playback. The main area displays a table of blocks:

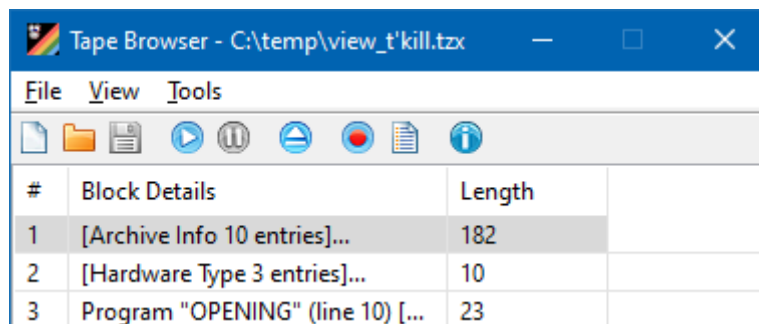
#	Block Details	Length
1	Program "sidea" (line 10)	19
2	[175 bytes of data]	175
3	Program "sidea" (line 10)	19
4	[175 bytes of data]	175
5	Program "sidea" (line 10)	19
6	[175 bytes of data]	175
7	Program "sidea" (line 10)	19
8	[175 bytes of data]	175
9	Program "sidea" (line 10)	19
10	[122 bytes of data]	122
11	Bytes "rainbow" (addr 16384)	19
12	[6914 bytes of data]	6914

At the bottom of the window, it shows 'Stopped' and 'Block 1 / 86'.

You can see the five instances of the program “sidea” which auto-runs at line 10, displays “VOLUME SET CORRECTLY”, waits 3 seconds then load the next program on the tape, followed by the rainbow loading screen “rainbow”.

Note that the block sizes included the two bytes of block length, which is why “rainbow” is showing 6914 bytes rather than the expected 6912 bytes – the size of the Spectrum’s screen memory. An option to exclude the two block size bytes will be added in a future Inkspector release. *Slacker! - Tape ed.*

The View → Additional Block Data expands data that would take up multiple lines, for example as per Archive and Hardware Type blocks used in .tzx files. As shown in this example:



The screenshot shows the 'Tape Browser' application window with the file 'C:\temp\view\_t\kill.tzx' open. The window has a menu bar (File, View, Tools) and a toolbar. The main area displays a table of blocks with expanded details:

#	Block Details	Length
1	[Archive Info 10 entries]...	182
2	[Hardware Type 3 entries]...	10
3	Program "OPENING" (line 10) [...]	23

See how the first two Block Details columns each end with an ellipsis, indicating additional information is available but currently not visible. Selecting the Additional Block Data menu item or toolbar button results in the following being displayed:



#	Block Details
1	[Archive Info 10 entries]
1	Title: A View To A Kill
1	Publisher: Stonesoft Ltd/Domark Ltd
1	Author: Tony Knight, Daryl Bowers, Gary Burfield Wallis.
1	Year: 1985
1	Language: English
1	Type: Game/Arcade
1	Price: N/A
1	Protection: None
1	Origin: Now Games 3 (Virgin)
1	Comment: TZXed by Andrew Barker
2	[Hardware Type 3 entries]
2	type 0 (Computers), id 1 (ZX Spectrum 48k, Plus) runs on this machine
2	type 4 (Joysticks), id 0 (Kempston) runs on this machine
2	type 4 (Joysticks), id 1 (Cursor, Protek, AGF) runs on this machine
3	Program "OPENING" (line 10) [Standard Speed Data Block. Delay = 1390ms]

## Creating A New Tape

To create a new tape, click on the Create New Tape toolbar button followed by Record (if there is no tape inserted, or it's already blank, you can click just on Record as a shortcut). The status bar should show "Recording". Depending on the machine that's currently running you can spool one of the following set of commands to demonstrate saving to tape and seeing the tape blocks generated as a result:

### ZX Spectrum

```
10 PRINT INK 5;PAPER 1;"Save demonstration"
20 SAVE "piccy"SCREEN$
run
```

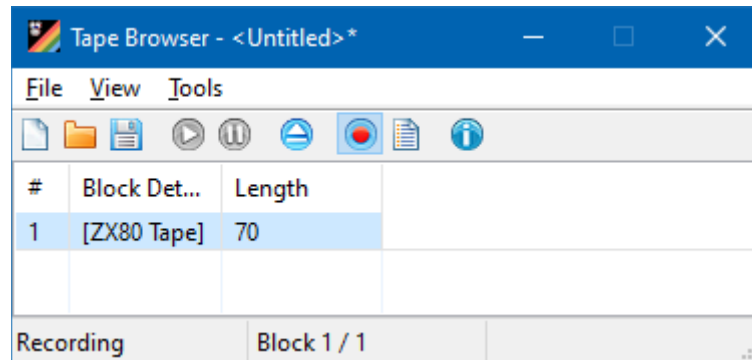
Then press a key as prompted.

#	Block Details	Length
1	Bytes "piccy" (addr 16384)	19
2	[6914 bytes of data]	6914

Recording      Block 2 / 2

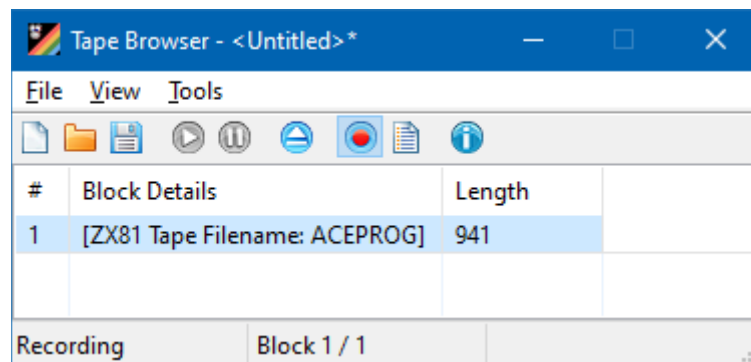
## ZX80

```
10 PRINT "ZX80 SAVE DEMONSTRATION"  
SAVE
```



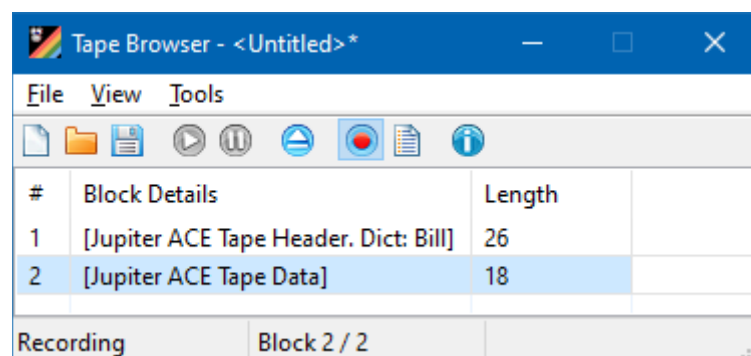
## ZX81

```
10 PRINT "ZX81 SAVE DEMONSTRATION"  
SAVE "ACEPROG"
```



## Jupiter ACE

```
: BILL VLIST VLIST ;  
SAVE Bill
```



With your newly created tape, you can then use File → Save As which will allow you to save it to a tape image format supported by the machine it was created on.

At this point, for the ZX Spectrum, ZX80 and ZX81 you can use Tools → View Programs On Tape which demonstrates how BASIC programs contained in tape blocks can be displayed without loading them.

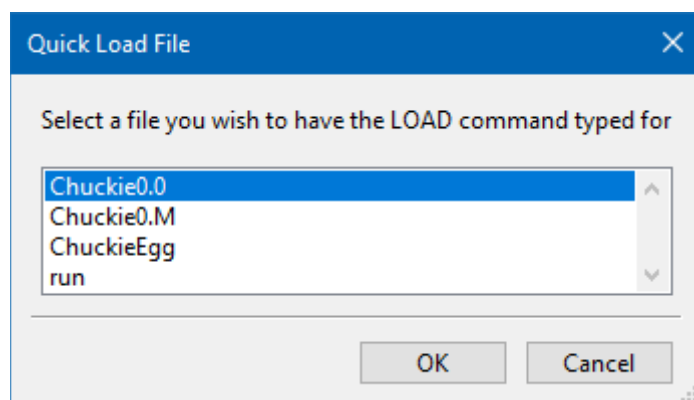
## Microdrives

The Microdrive Control panel is where existing Microdrive cartridges may be inserted, created or examine using the Microdrive Map. If the current machine does not have a ZX Interface 1 attached, the window is still available but all menu items and toolbars will be greyed out and unavailable until one is attached.

Loading an existing cartridge file can be done in the usual ways via File → Insert Existing Cartridge File menu item or toolbar button. As with the main window, it also has a Recent Microdrive Cartridges menu that keeps track of the last ten cartridge image files loaded or saved.

Selecting the Tools → Quick Load menu item or toolbar button brings up a window that allows you to select one of the files present from the currently selected Microdrive. Pressing OK on this window has Inkspector auto-type the BASIC LOAD command required to load in the selected file.

For example, with a Chuckie Egg cartridge inserted, selecting Quick Load shows:



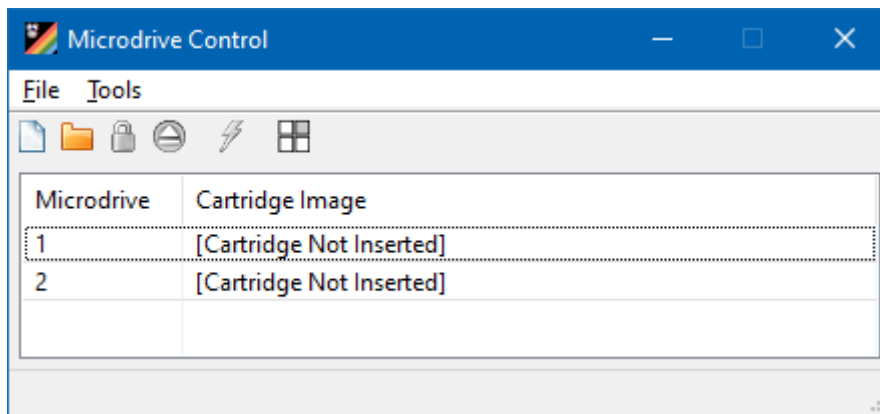
And if we select “run”, the following is typed in:

```
LOAD *"m"; 1; "run"L
```

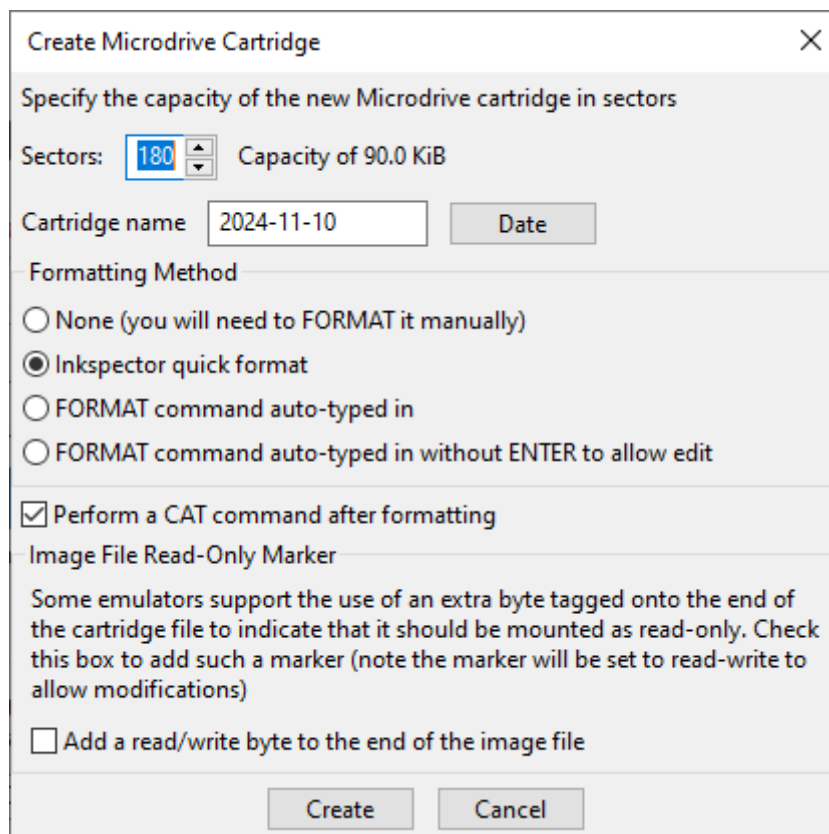
If you hold down CTRL as you press OK, the line is typed without pressing ENTER at the end, to give you opportunity to manually edit the line.

Right then. To make things *slightly* more interesting, let's create our own cartridge and demonstrate the features that way, rather than just opening an existing .mdr cartridge file...\*yawn\*

While running a 48K Spectrum with a ZX Interface 1 attached, open the Microdrive Control window by pressing Alt-R (or you could use View → Microdrives on the main window's menu). We can see we have two Microdrive units attached to the Interface 1, both without cartridges inserted.



To create a new cartridge, click on Create New Cartridge on the toolbar or File menu.



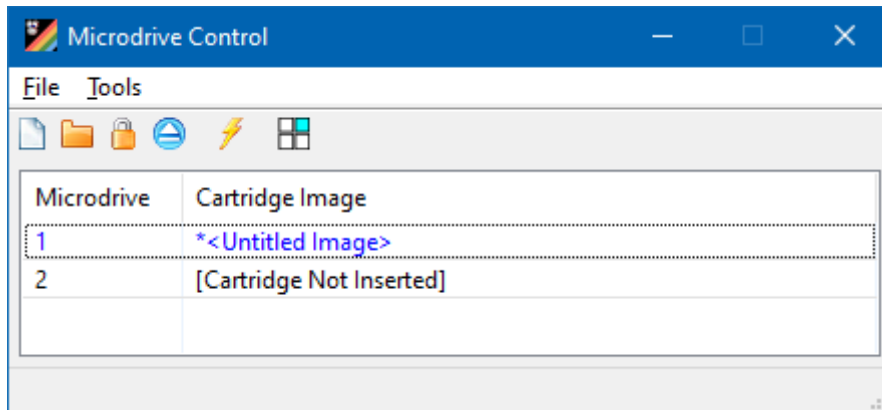
The default capacity of a new cartridge is 90Kb (180 sectors), but you can change it as you wish, up to a maximum capacity of 127 Kb (254 sectors).

The next option is to specify the cartridge's name, which may be up to 10 characters long. The default name is the current date in yyyy-mm-dd format. You can press the Date button to reset the name back to the current date.

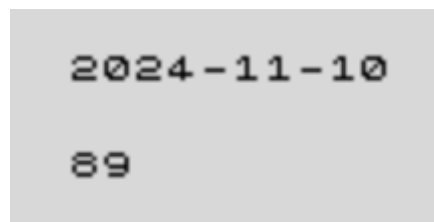
The next group of options select whether or how to format the new image. I would suggest using the default of Inkspector quick format, as saves you from having to issue a `FORMAT *”m”;1;”2022-08-07”` afterwards or watch the border flash as the `FORMAT` command is executed. Unless you like that, in which case, knock your pipe out. I’m not here to judge.

The final option is whether to add a read-only marker to the end of the image file as beautifully described above that checkbox.

Once you're happy with these options, press Create. You should then see something along the lines of this in the Microdrive Control, confirming the image file has been created and inserted into the Microdrive unit. The asterisk and blue colouring indicating that the changes to the image haven't yet been saved to a file.



If you type CAT 1 into the Spectrum (I suggest using [Keyboard Assist](#) here to save your fingers, especially when moving on to the lengthier Microdrive commands with the fruity syntax) you'll be rewarded with a catalogue showing the formatted volume and capacity:

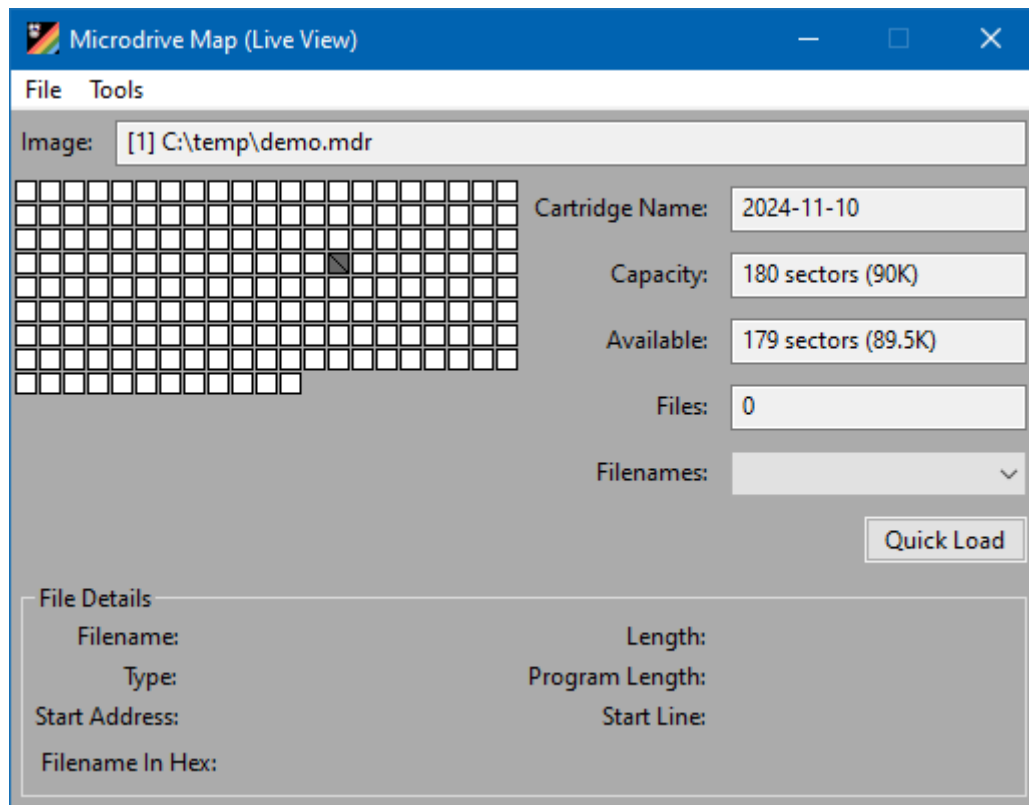


Hmm...just hang on a cotton picking minute, didn't we set the capacity of the new cartridge to 90Kb? Why is it showing 89?

Well, yes, we did. If only there was a feature that allowed us to see what's going on! (© 2025 Segues-R-Us)...

## Microdrive Map

Clicking on the Microdrive Control's 'Show Cartridge Map' toolbar button or File → View Microdrive Map will show the following:



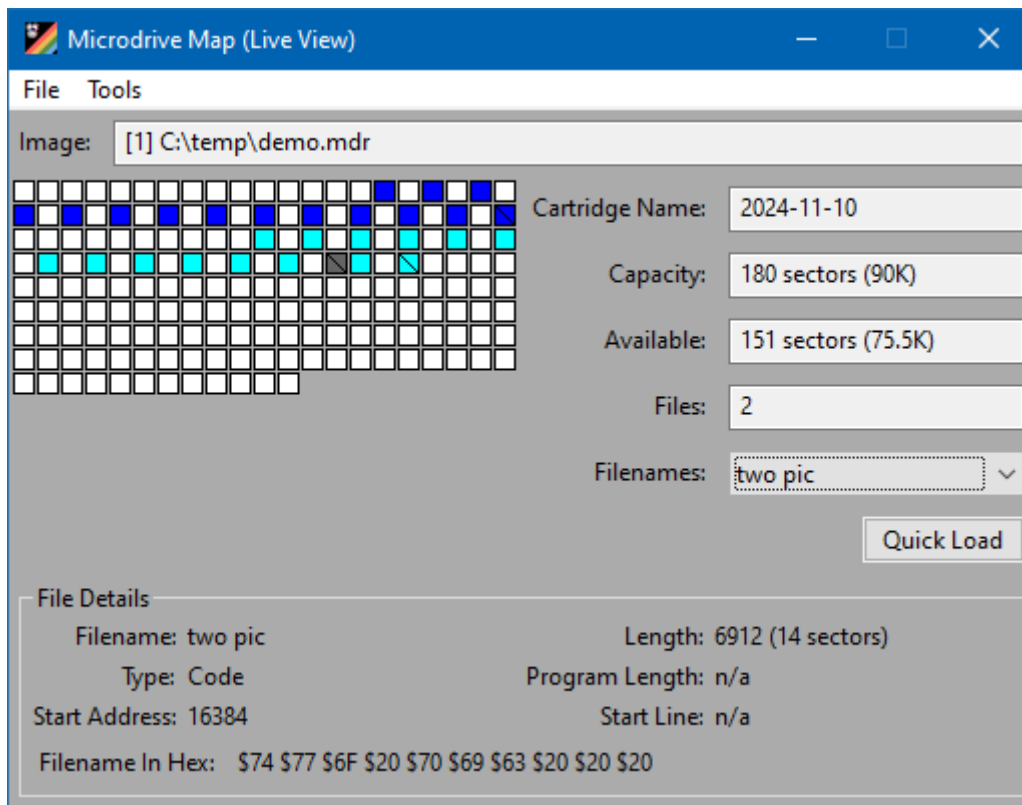
The grid area is divided into colour coded sector units:

Colour	Meaning
White	Currently unused
Blue	Occupied by a file
Grey	Occupied by a print file
Cyan	Indicating the currently selected file from the Filenames drop-down list
Red	Unformatted

Pressing the “Quick Load” button auto-types the BASIC LOAD command required to load the file currently selected from the Filenames drop down list.

Sectors with the EOF (end of file) marker set have a diagonal line through them.

So, back the mystery of the 89KB cartridge. We can see that one sector has been used, reducing the count of available sectors down to 179, and since the Interface 1 ROM shows the number of KB available by doing an integer divide of the available sectors by two, that’s how the Interface 1 ROM arrives at 89KB being reported as available on the cartridge. Alles ist klar!



Live View (File → Live View) ensures the display is kept up to date every time the contents of the current Microdrive cartridge changes.

## Sector List

The Microdrive Sector list (accessible from Tools → Sector List menu item on the Microdrive Control or the Microdrive Map windows) displays all the sectors for the cartridge in the selected Microdrive.

	Sector	In Use	Header Flag	H-Chk	Record Flag	Rec. Num	R-Chk	Filename	Data Len.	D-Chk
13	242	Yes	\$01	\$C6	\$06	\$02	\$24	Chuckie0.0	206	\$CE
14	241	No	\$01	\$C5	\$00	\$00	\$00		0	\$00
15	240	No	\$01	\$C4	\$00	\$00	\$00		0	\$00
16	239	No	\$01	\$C3	\$00	\$00	\$00		0	\$00
17	238	Yes	\$01	\$C2	\$04	\$00	\$70	Chuckie0.M	512	\$57
18	237	No	\$01	\$C1	\$00	\$00	\$00		0	\$00
19	236	Yes	\$01	\$C0	\$04	\$01	\$71	Chuckie0.M	512	\$A8
20	235	No	\$01	\$BF	\$00	\$00	\$00		0	\$00

☐ Show In-Use Sectors Only

Summary

Header Checksum Errors: 0

Record Checksum Errors: 0

Data Checksum Errors: 0

Read / Write Tag Present In File: Yes - read / write

OK

The columns show the values for each sector. The values in H-chk (header checksum), R-Chk (record checksum) and D-Chk (data checksum) are shown in green when the checksum matches the

data they represent, or red when the checksums do not match. The exception being D-Chk, which remains in black when its value is not in use by the system.

Checking 'Show In-use Sectors Only' hides the sectors that are not in use

Microdrive 1 Cartridge Sectors										
	Sector	In Use	Header Flag	H-Chk	Record Flag	Rec. Num	R-Chk	Filename	Data Len.	D-Chk
1	254	Yes	\$01	\$D2	\$06	\$00	\$6C	run	47	\$3A
2	250	Yes	\$01	\$CE	\$06	\$00	\$00	ChuckieEgg	294	\$DB
3	246	Yes	\$01	\$CA	\$04	\$00	\$53	Chuckie0.0	512	\$EF
4	244	Yes	\$01	\$C8	\$04	\$01	\$54	Chuckie0.0	512	\$79
5	242	Yes	\$01	\$C6	\$06	\$02	\$24	Chuckie0.0	206	\$CE
6	238	Yes	\$01	\$C2	\$04	\$00	\$70	Chuckie0.M	512	\$57
7	236	Yes	\$01	\$C0	\$04	\$01	\$71	Chuckie0.M	512	\$A8
8	234	Yes	\$01	\$BE	\$04	\$02	\$72	Chuckie0.M	512	\$83

☒ Show In-Use Sectors Only

The only sector information not shown on this window is the 512 bytes of sector data, however clicking on 'View Sector Data' will show a hex dump of the data for the current sector

Microdrive 1 Cartridge Sectors										
	Sector	In Use	Header Flag	H-Chk	Record Flag	Rec. Num	R-Chk	Filename	Data Len.	D-Chk
1	254	Yes	\$01	\$D2	\$06	\$00	\$6C	run	47	\$3A
2	250	Yes	\$01	\$CE	\$06	\$00	\$00	ChuckieEgg	294	\$DB
<div>View Sector #4 (formatted as #250)</div> <div>0000 00 1D 01 05 5D 1D 01 0A 00 00 0A 85 00 FD B0 22 ....]....." 0010 32 34 39 31 31 22 3A E7 B0 22 30 22 3A F9 C0 B0 24911":.."0":... 0020 22 32 33 39 35 35 22 3A F1 61 24 3D 22 43 68 75 "23955":..a\$="Chu 0030 63 6B 69 65 30 2E 22 3A F1 64 3D BE B0 22 32 33 ckie0.:.d=..23 0040 37 36 36 22 3A EB 69 3D B0 22 30 22 CC B0 22 30 766":.i="0".."0 0050 22 3A EF 2A 22 6D 22 3B 64 3B 61 24 2B C1 69 AF ".*"m";d;a\$+.i. 0060 3A F9 C0 B0 22 33 32 31 37 39 22 3A F3 69 3A EF :...32179":.i:. 0070 2A 22 6D 22 3B 64 3B 61 24 2B 22 4D 22 AF 3A F2 *"m";d;a\$+"M"... 0080 B0 22 30 30 31 22 3A F9 C0 B0 22 32 33 39 39 :."001":..."23999 0090 22 0D 27 0F 99 00 EA F3 2A 3D 5C 23 36 13 2B 36 ".'. ....*="#6.+6 00A0 03 2B 36 1B 2B 36 76 2B 36 00 2B 36 51 F9 FD CB .+6.+6v+6.+6Q... 00B0 01 A6 3E 00 32 8D 5C CD AF 0D 3E 10 01 FD 7F ED ...&gt;.2.\&gt;..... 00C0 79 FB C9 F3 21 23 D1 11 00 5B 01 36 00 ED B0 31 y...!#...[.6...1 00D0 05 5E D9 01 FD FF AF E1 ED 79 3C 06 BF ED 69 06 .^.....y&lt;...i. 00E0 FF ED 79 3C 06 BF ED 61 FE 10 06 FF 20 E9 3E 00 .y&lt;...a.....&gt;. 00F0 ED 79 C1 D1 E1 D9 DD E1 FD E1 08 F1 08 3E B2 ED .y.....&gt;.. 0100 47 ED 5E 31 36 5B C3 02 5B 00 00 00 00 00 00 00 G.^16[...[..... 0110 00 00 00 00 00 00 00 00 00 00 21 16 9B 36 58 27 E8 .....!..6X'. 0120 CB 3A 5C 62 22 0D 00 00 00 00 00 00 00 00 00 ..\b"..... 0130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....</div>										

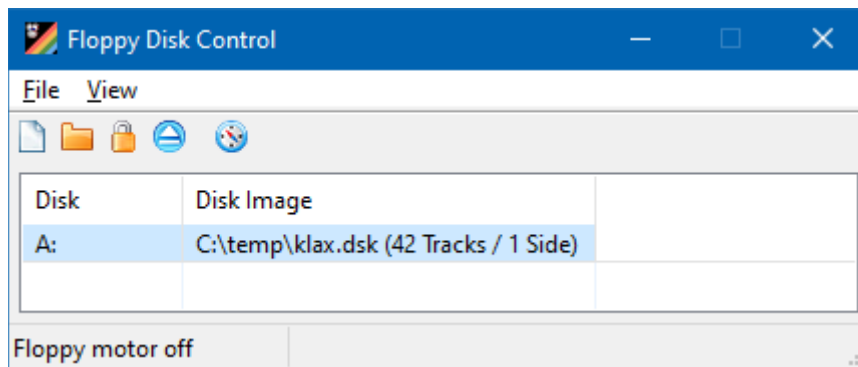
☒ Show In-Use Sectors Only

## Disks

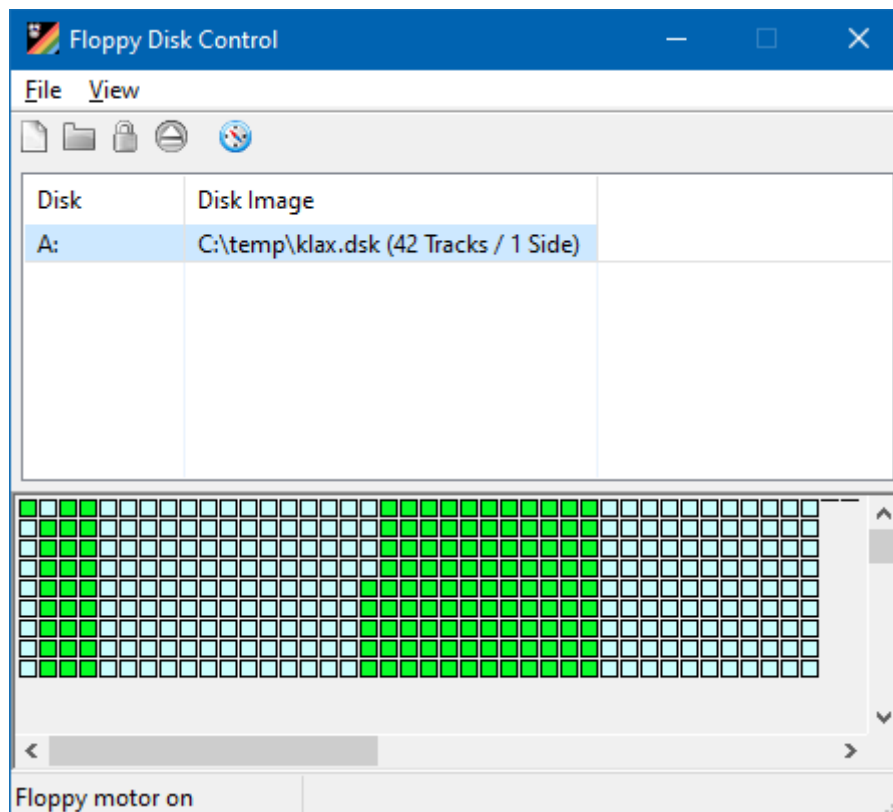
When running a Spectrum +3 you can insert, format or eject a floppy disk image.

Left to right, the toolbar buttons perform: create new disk image, open an existing disk image, lock the current disk image (prevents any writes, as though the write-protect tab is set), eject disk and finally show disk map.





Pressing the compass toolbar button on the right or selecting View Map from the View menu shows the disk map, showing all the sectors on the disk in the selected drive:



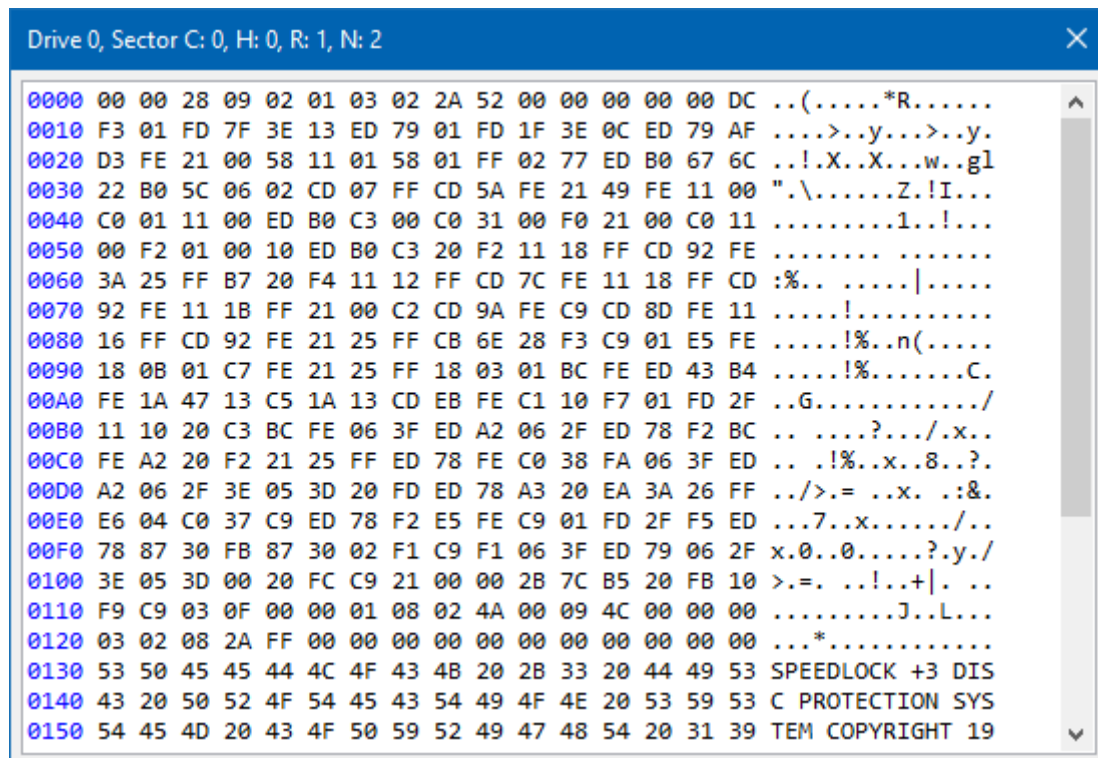
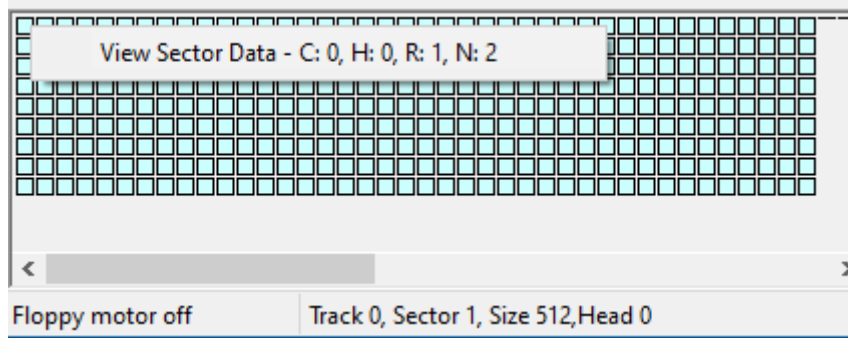
Hovering the mouse cursor over a sector box shows its details on the status bar. The colour coding indicates how recently a sector has been read (green) or written to (red), with the colours slowly fading back to the default 'cold' colour of cyan after ten seconds (this value may be changed on the Floppy Disks options page). Sectors marked as 'deleted' have a diagonal line through them. Sectors that have multiple copies of data stored for them (for weak sector support) are shown in a pale yellow colour. An example of these is shown below:



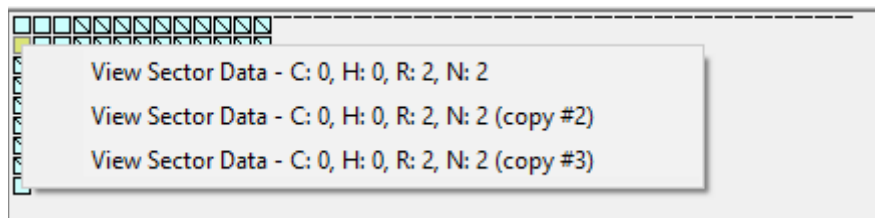
The View → View Disk Map Scaled option shows the sector boxes scaled to their reported sizes. The above disk map looks as follows with it enabled:



Right-clicking a sector box brings up a menu option to view the sector data



Right-clicking on a sector that has multiple copies of it available (as indicated by the pale yellow colouring) gives you the option to select which copy to view:

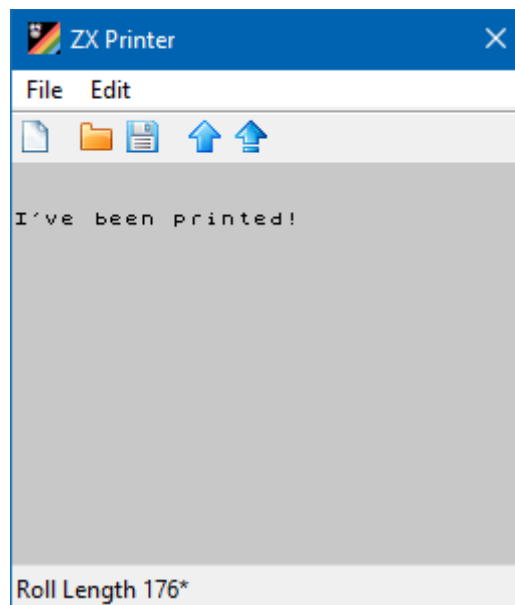


## ZX Printer

When a ZX printer is attached, you can view what is printed to it by opening this window. With it open, spooling the following program

```
10 PRINT "I've been printed!"
20 COPY
run
```

Results in:



From the File menu and toolbar you can clear the current paper roll, save it as a .zxpri paper roll file, load one or (from the File menu) save as a .png, .bmp or .pbm (in text format) image file. The solid blue arrow is a single line paper feed. The one to the right with the gap is a character paper feed (8 lines).

## Debugger

The Inkspector debugger supports the usual features such as single stepping, stepping over, stepping out (returning to the calling function), executing until a certain line, etc. In addition, there is a powerful [breakpoint](#) system and disassembler available (does anyone know of, or remember, dZ80? It's still earning its keep under the hood here providing the Z80 disassembly).

**InkBrag:** Unlike some emulators, the main Inkspector window remains accessible while the

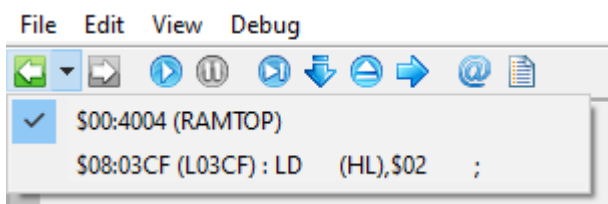
debugger is open, so that all the usual functionality continues to be available. For example, to load a snapshot, just go to the usual File → Load Supported File menu on the main window. You don't have to do it from a special or duplicated menu on the debugger itself. And you can continue editing Snippets, changing the machine type or whatever your heart desires.

### InkTip

If you open the debugger window before loading a snapshot, the snapshot will be loaded in a paused state and the debugger will show the instruction at the point the snapshot was saved.

## Toolbar

The debugger toolbar is shown below, with the navigation history arrow clicked on, showing the current history:



	Navigate back through the history. As shown above, clicking this would take the debugger back to address \$03CF, which contains a 'LD (HL), \$02' instruction. The history is maintained automatically and added to when you navigate within the debugger to an address roughly 20 bytes away from the current one. As shown above, clicking the down arrow shows the current history.
	Navigate forwards through the history. This is greyed out (as shown above) when the navigation is currently at the end of the history.
	Run / unpause the emulator (if you can call it an emulator)
	Pause the emulator
	Step into the next instruction (F11)
	Step over the current instruction. e.g. if the instruction is a 'call', execute the call and break at the instruction after.
	Step out. Execute the current function and break when it returns to the caller.
	Run to here. Breaks when the Z80 reaches the instruction at the cursor.
	Toggles the display of symbols
	Toggles the display of listings and source code

## Edit Menu

Copy Cursor Address copies the address where the cursor is currently placed.

Copy On-Screen-Disassembly copies the on-screen disassembly to the clipboard, albeit without any symbol names.

Find, Find and Replace and Write memory opens up the [Find Memory](#) dialog using the debugger cursor's address as a starting point.

You can also summon the [Poke Memory](#) dialog from this menu.

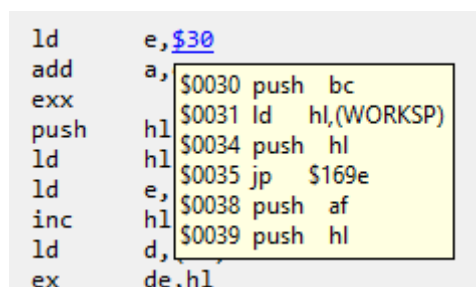
## View Menu

Ticking “Full Address” shows the full address in the debugger's disassembly. That is, it shows which ROM or RAM page the address belongs too in addition to the 16-bit address that the Z80 can see. See the [Disassembly Area](#) for the full list of possible ROM and RAM pages.

“View Opcodes” controls whether the display of the Z80 op-codes is shown immediately to the right of the address on each line of disassembly.

“Symbols” and “Source Code” allow overriding of the display of either when symbols or source [are available](#).

“Peek Disassembly” shows a tooltip containing a small disassembly starting at the address of the hyperlink hovered over, e.g. when the mouse cursor is hovered over the “\$30” hyperlink:



“Z80 Registers” displays the [Z80 Registers](#) window.

“Goto Address” and “Goto Symbol” change the debugger's current address.

Ticking “Stack and Breakpoints” enables or disables the display of the stack dump and breakpoint list display. This can be useful if you want to maximise the area given to the debugger's disassembly area.

“Hexadecimal Display” toggles the global Inkspector hexadecimal number setting, immediately affecting all other open Inkspector windows.

“Navigate Forward” and “Navigate Backward” are menu shortcuts that provide the same functionality as the navigate buttons on the [debugger toolbar](#).

“Debugger Options” opens up the Options screen at the Debugger page, so you can quickly change debugger settings.

“Clear Single Step Times” clears any [t-state timings](#) that are currently displayed.

“Refresh” just causes the debugger screen to be redrawn. This shouldn’t ever be necessary as it should always be up-to-date.

## Debug Menu

Since the File, Edit and View menu items are fairly self-explanatory, I’ll focus on the Debug menu which contains some interesting functionality.

The first half dozen or so items mirror the stepping and execution functionality available from the toolbar.

Add Breakpoint allows a new breakpoint to be created, exactly as Add Breakpoint on the main window’s Tools → Add Breakpoint menu does.

Add Breakpoint At Cursor is similar to Add Breakpoint, except that the breakpoint’s address is set to the address of the cursor in the disassembly window.

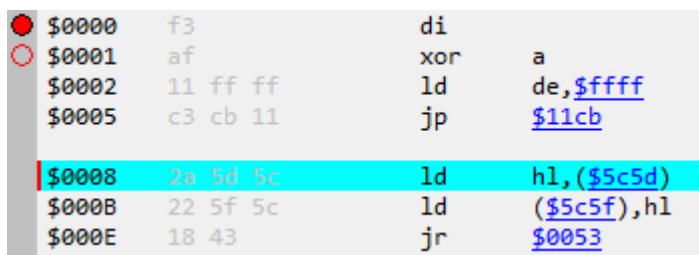
Toggle Breakpoint At Cursor toggles an Op-code Read breakpoint at the address of the cursor. ‘Yer basic breakpoint as it were.

Break on Branch and Break on no Branch are only available when the cursor is on a conditional instruction. They create Op-code Read breakpoints that break only when the condition is met (Break on Branch) or not met (Break on no Branch). For example, setting Break on Branch for a ‘ret z’ instruction will only break when the instruction is executed and the zero flag is set, and Break on no Branch would break when the zero flag is reset. You can also switch between branch/no branch breakpoints by clicking on the relevant line and re-selecting one of the two Break on... menu items.

The remainder of this menu’s items are also self-explanatory.

## Disassembly Area

Most of the debugger’s window is dedicated to showing the Z80 instructions. The display is arranged in the following columns, from left to right:



\$0000	f3	di	
\$0001	af	xor	a
\$0002	11 ff ff	ld	de,\$ffff
\$0005	c3 cb 11	jp	\$11cb
\$0008	2a 5d 5c	ld	hl,(\$5c5d)
\$000B	22 5f 5c	ld	(\$5c5f),hl
\$000E	18 43	jr	\$0053

Leftmost is the breakpoint column, show in a dark grey strip. The example above shows a solid red circle at address \$0000, indicating there’s an active Op-code read breakpoint at that address. The hollow circle on the line below at address \$0001 indicates the same type of breakpoint (red outline) that has been disabled. Different colours indicate different memory breakpoint types:

Colour	Breakpoint Type
Red	Op-code read
Blue	Memory read

Yellow	Memory write
Green	Memory read/write

The next column shows the line's address. By default, a 16-bit address is shown, but it's possible to show the ROM or RAM page the address is occupying (even for machines that do not support ROM or RAM paging) by selecting View → Full Address. Selecting this turns the above screen into this:

ROM00:\$0000	f3	di	
ROM00:\$0001	af	xor	a
ROM00:\$0002	11 ff ff	ld	de,\$ffff
ROM00:\$0005	c3 cb 11	jp	\$11cb
ROM00:\$0008	2a 5d 5c	ld	hl,(\$5c5d)
ROM00:\$000B	22 5f 5c	ld	(\$5c5f),hl
ROM00:\$000E	18 43	jr	\$0053

The five characters that have appeared in front of the 16-bit address (when View → Full Address is enabled) will be one of the entries as shown in the table [here](#). If the address is unmapped, UNMAP will be shown.

After the optional ROM/RAM number and the mandatory address, the bytes that make up the Z80 instruction are shown in light grey as not to distract. If you don't want to see them at all, select View → Opcodes to toggle between hiding and showing them.

The next column along is the Z80 instruction for the address. Values underlined and coloured in blue are hyperlinks that will take the debugger to that address when clicked on. Additionally, when View → Peek Disassembly is checked, a short disassembly of the address at the value is shown as a tooltip to provide a quick way to see what lies there, as shown in the picture below (not that you can see it, but the mouse cursor is hovering over the address part of the 'jp \$11cb' instruction at \$0005)

ROM00:\$0000	di	
ROM00:\$0001	xor	a
ROM00:\$0002	ld	de,\$ffff
ROM00:\$0005	jp	\$11cb
ROM00:\$0008	ld	hl, \$11CB ld b,a
ROM00:\$000B	ld	(\$5 \$11CC ld a,\$07
ROM00:\$000E	jr	\$00 \$11CE out (\$fe),a
ROM00:\$0010	jp	\$15 \$11D0 ld a,\$3f
ROM00:\$0013	rst	\$38 \$11D2 ld i,a
		\$11D4 nop

The blank lines in the disassembly are inserted after an instruction that has the ability to branch such as JumpPs and RETurns (including conditional ones) to make it easier to read. You can turn off this spacing from the [Options → Debugger](#) page.

The Z80's current address (PC) is indicated by the entire line being coloured with cyan. The cursor you can see at the start of address \$0008 above is used for navigating and scrolling. At any time you can press the SPACE bar or select Debug → Show Next Instruction to take the display back to the cyan line / current Z80 PC address.

Once you start single stepping through code, you will see that numbers displayed in red brackets are shown. These are the number of t-states taken to execute each instruction. For example, single stepping through the first three instructions above, results in:

●	ROM00:\$0000	(4)	di	
○	ROM00:\$0001	(4)	xor	a
	ROM00:\$0002	(10)	ld	de,\$ffff
	ROM00:\$0005		jp	\$11cb

You can clear the t-states timings (thus removing the red numbers until more single-stepping or stepping over is performed) by selecting View → Clear Single Step Times. You can have the numbers shown after the Z80 instruction instead of before, or turn this feature off completely from the [Options → Debugger](#) page.

Right-clicking on the disassembly area will display a popup menu giving access to some of functionality on the Debugger's own menu. NB where the popup menu item operates on an address (such as 'Set Next Instruction') it will be taken from the blinking cursor's current position, not the mouse position.

## Call Stack Area

\$7FDC -> \$0000 (L0000)
\$7FDE -> \$0000 (L0000)
\$7FE0 -> \$C10D
\$7FE2 -> \$C104
\$7FE4 -> \$028B (L028B)
\$7FE6 -> \$4119
\$7FE8 -> \$00FE
\$7FEA -> \$0001

This area is a simple dump of the values currently on the stack and the addresses they point to, along with any matching symbol names in brackets. The line in blue represents the start of the stack (i.e. the value of SP, in this case \$7FE4).

Double clicking on any line navigates to the address the stack entry points to.

## Breakpoints Area

<input checked="" type="checkbox"/> [1] NMI Taken
<input checked="" type="checkbox"/> [2] Op-Code Read \$03E9 (ZX81 ROM)
<input checked="" type="checkbox"/> [3] Op-Code Read \$03E0 (ZX81 ROM). When @nz

This area displays all current breakpoints. Active ones have a tick in their box, disabled ones do not. You can click on the box to toggle their state.

Double-clicking on a breakpoint edits it in the [breakpoint window](#).

## Expression Box

<input type="text"/>	=	Result
----------------------	---	--------



This area allows you to enter an expression in the box on the left to see the result on the right, similar to the [Calculator](#). Additionally it has limited abilities to assign values to registers as we shall see. The format expected to be entered in the input area is <expression> with one or more optional settings by separating with a comma.

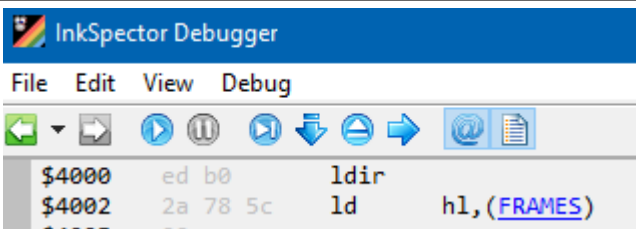

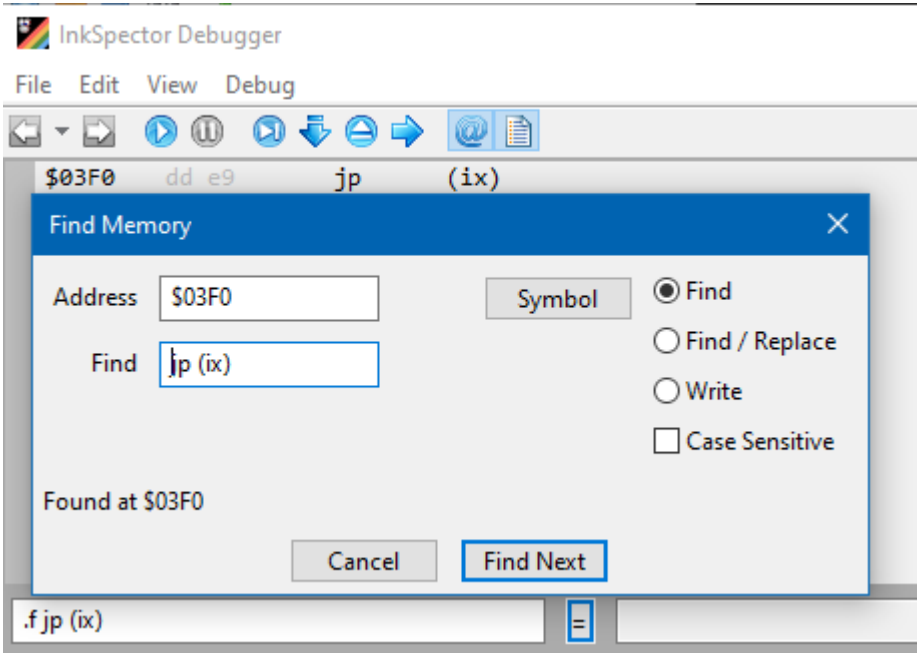
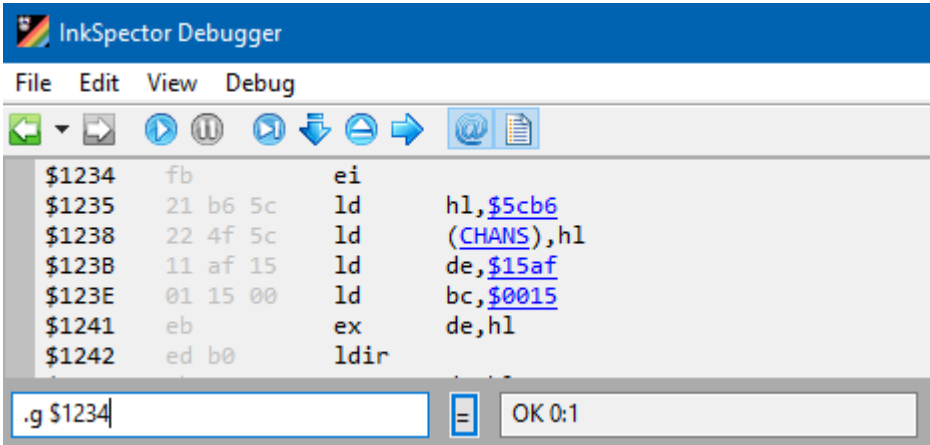
Option	Effect
b[n]	Shows the result in binary, optionally specifying the minimum number of digits to be shown, e.g. 1234,b or 1234,b24
B	Shows the result in fixed 16-digit binary
d	Show the result in decimal (for example, if Inkspector is currently showing values in hexadecimal)
x	Show the result in hexadecimal
m[p][f]	Sends the result to page number p on the <a href="#">memory window</a> (or page 1 if not specified), optionally setting the “Follow” box if ‘f’ is specified at the end. Examples: “1234,m” (set page 1 of the memory window to address 1234). “1234,m2” (set page 2 of the memory window to address 1234). “1234,m3f” (set page 3 of the memory window to address 1234 and check its ‘follow’ box).  NB this option has been superseded by the .m command below.

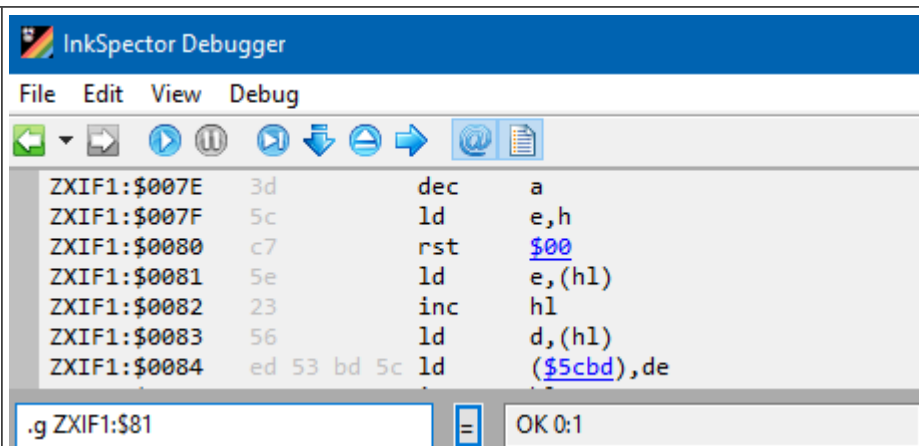
1234	=	1234 = 1234
1234,b	=	1234 = 10011010010
de + 10	=	de + 10 = \$2f79
hl=FRAMES	=	FRAMES = \$5c78 -> hl
pc = -1+\$5c78/FRAMES	=	-1+\$5c78/FRAMES = \$0 -> pc

(snigger – mischief ed.)

There is one exception. Starting the input with a single dot specifies a command. The recognised commands are:

Command	Function
.a <i>param</i> .	Assembles <i>param</i> . to the current cursor position. This command may only be used when the machine is paused.  <div> <input type="text" value=".a ldir"/> <input type="button" value="="/> <div>Assembled 2 bytes to RAM05:\$4000</div> </div> <div> <input type="text" value=".a ld hl,(FRAMES)"/> <input type="button" value="="/> <div>Assembled 3 bytes to RAM05:\$4002</div> </div> <p>Which results in the debugger showing</p>

	
<code>.l param.</code>	<p>Lists the Z80 opcodes from the assembly of <i>param</i>, intended as a Z80 op-code reference.</p> 
<code>.f param.</code>	<p>Opens the Find Memory dialog box, initiating the search with <i>param</i>.</p> 
<code>.g param.</code>	<p>Go to the address specified by <i>param</i>. It may contain a ROM or RAM page number.</p>  <p>To go to address \$0081 in the ZX Interface 1 ROM:</p>

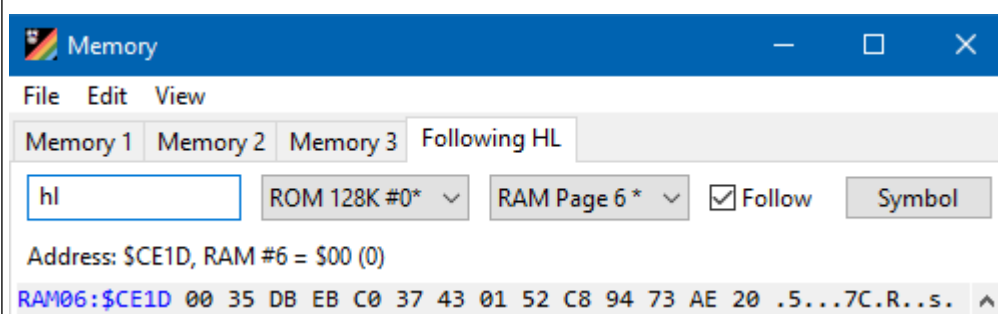


See [inks.get\\_full\\_address\\_from\\_str\(\)](#) for all supported ROM and RAM page names.

**.m *param***

Sends *param* to the Memory window. The format of *param* is:  
*address*[, *mem-tab*, *follow*, *tab-name*]

<i>address</i>	Address to set the memory window tab to
<i>mem-tab</i> (optional)	Which memory tab to use (1-4, 0 = use the current tab if window is open)
<i>follow</i> (optional)	0 = don't follow the memory address 1 = follow the memory address
<i>tab-name</i> (optional)	Sets the memory tab's name. If not specified, <i>addr</i> is used



**.w *param***

Sends *param* to the Watch window

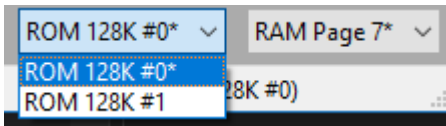
**.W *param*.**

Displays *param* formatted as per the [Watch](#) formatters.



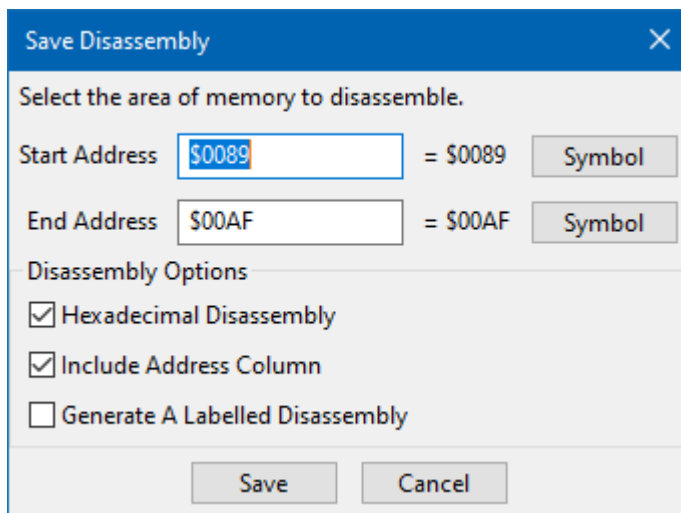
## ROM and RAM Selectors

These selectors allow you to override the ROM and RAM pages currently being viewed by the debugger. Items with an asterisk by them indicate they're currently paged in.



## Disassembler

You can save a disassembly to a file by selecting File → Save Disassembly



The default start and end addresses are those of the disassembly currently being shown on the debugger window.

Checking *Include Address Column* includes the current address at the start of each line of the disassembly.

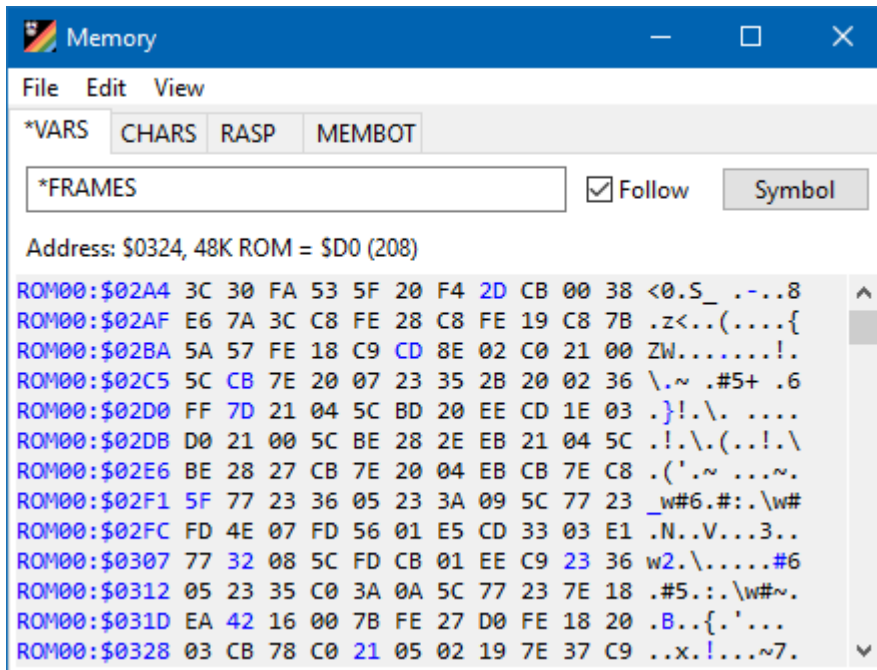
Checking *Generate A Labelled Disassembly* performs a scan of the code and generates labels for addresses that are referenced.

## Memory

The Memory window displays the contents of the emulated system's memory at the address entered into the address box. There are four memory tabs that can be configured separately. Wait! Before you jump to the next chapter due to tedium, this window does have a couple of tricks up its sleeves (do windows have sleeves?)...

For example, using our ubiquitous friend the FRAMES system variable and the “contents of” \*[operator](#), we can have the memory window point to whatever value is being held in the FRAMES system variable (not the address of the variable itself). And when the “Follow” box is checked, it will automatically re-evaluate the address once a second and update the display if the resulting address changes (similarly, you could have the memory window follow the Z80's program counter around by entering “pc” as the address).

So having entered “\*FRAMES” into the address box, you should be rewarded with something that looks like this:



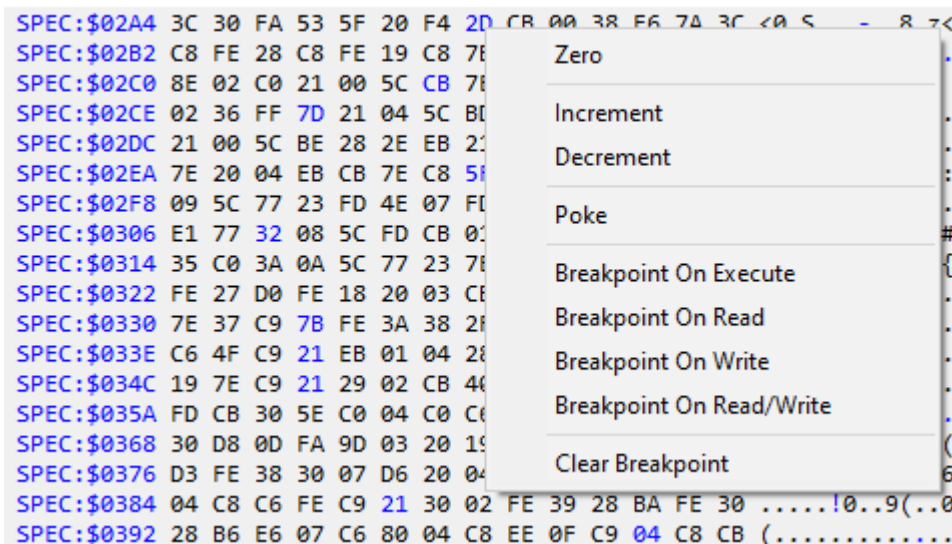
Obviously the chances of your address being the exact same as the above are pretty remote since FRAMES is incremented every 50<sup>th</sup> of a second, but you get the idea.

Memory bytes that are shown in blue indicate there is a symbol whose value corresponds to its address. If you click on the byte, the symbol will be shown. Because the machine is paused (indicated by the memory bytes being shown in black – they would be light grey if the machine was running), we can click on the blue “2D” on the first row, and the symbol name – L02AB – will be shown in brackets:

Address: \$02AB, 48K ROM (L02AB) = \$2D (45)  
ROM00:\$02A4 3C 30 FA 53 5F 20 F4 2D CB 00 38 E6 7A 3C <0.S\_ .-..8.z<

Right-clicking in the memory bytes area brings up a menu that allows you to modify it. In addition you can directly set a breakpoint for when it’s written to, read or executed from:

Address: \$02B9, 48K ROM = \$7B (123)

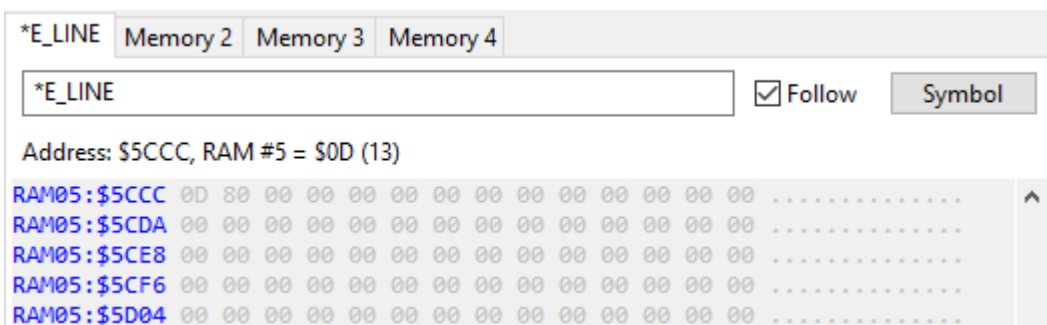


As with the debugger, memory breakpoints are shown [colour coded](#).

## Tracking Memory Changes

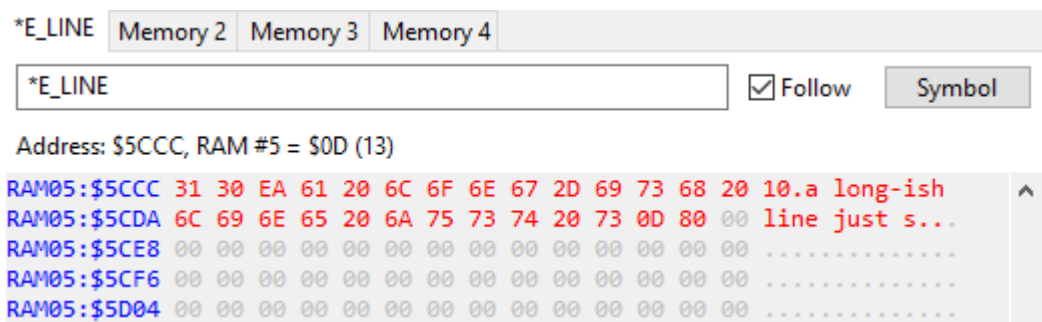
Sometimes it's useful to be able to look for changes being made to an area of memory. The memory window has a feature that makes doing this a little easier. Of course it does. *Making sure that both View → Track Changed Memory and View → Update While Running menu items are ticked*, enter \*E\_LINE in the address box and reboot a 48K Spectrum.

Once the Spectrum has rebooted and is displaying the 1982 copyright message, the memory window should look like this:



The memory window is now obediently displaying the memory pointed to by the E\_LINE system variable, which contains the address of the BASIC line being entered. If you copy the BASIC line below into the clipboard and use (from the main window) Tools → Spool Clipboard you should see the updated memory highlighted in red as the line is entered into the machine.

```
10 REM a long-ish line just so that you can see it being entered into the memory
pointed to by the E_LINE system variable. This line is just going on and on and
on and on and on and on...
```



By default, the changes to memory shown in red will clear in 5 seconds, but you can disable this by unticking View → Clear Changes After 5 Seconds. If you do this you may want to clear the changes manually now and again, using View → Clear Changed Memory.

## Find Memory

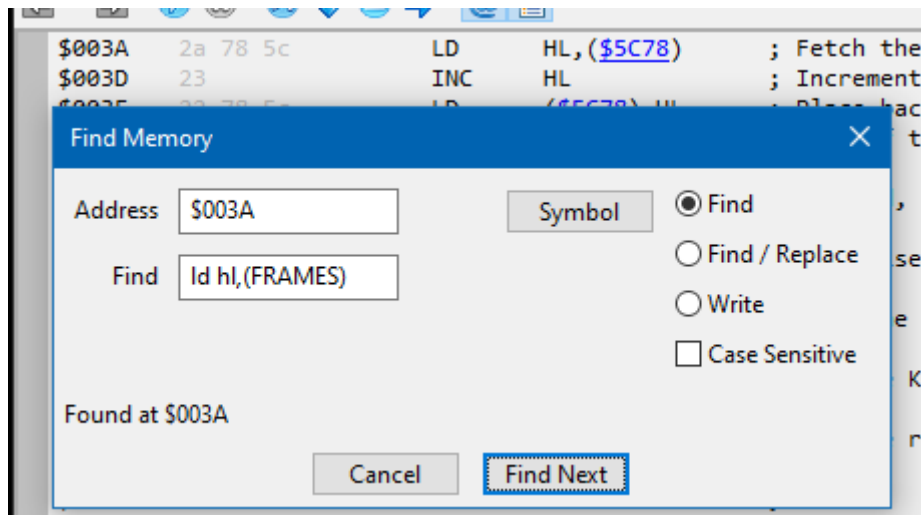
The find memory tool is available from the [debugger](#) and [memory](#) windows. As you’d expect, it allows you to find patterns of bytes in the current machine’s memory and sets the debugger or memory address accordingly. What you might not expect is just how waffle-y versatile it is (look, if I don’t sell Inkspector, who will?). This tool operates in one of three modes, each described below.

### Find

This simplest of the modes searches all of the machine’s memory starting from the value in the “Address” text box. Address may be specified as a simple number. But as with most Inkspector text entry boxes, an [expression](#) such as “\$1000+\$1024”, “FRAMES”, etc., is accepted.

The bytes to search for are specified in the “Find” text box. It may be a single value, or multiple ones separated with a comma. Each individual value is treat as an expression resulting in a byte value to search for. As a result, *each individual search value is clamped to 8-bits*, so searching for “hl” (the current Z80 value of the HL register pair) will only search for L. If you want to search for the two bytes represented by the current value of HL, specify “l,h”.

As well as being able to search for expressions and Z80 register values, you can also search for a single assembly instruction, e.g. “ld hl,\$4000”, “ld hl,(FRAMES)”, etc., as shown in this picture. Note how the debugger’s start address has automatically set to the address containing “ld hl, (FRAMES)”, FRAMES being a system variable with the value \$5C78 on a 48K Spectrum.

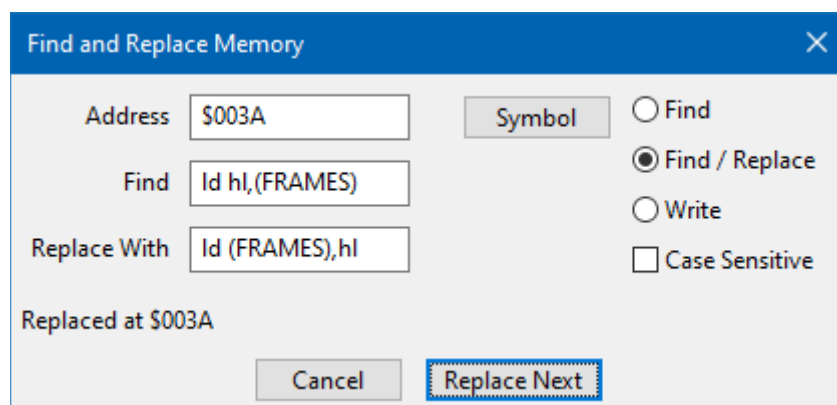


Example find values

Find Value	Bytes Searched For (Decimal)
1	1
1,2,3	1,2,3
"Life"	76,105,102,101
\$FF,\$80	255,128
ld hl,(FRAMES)	42,120,92
ldir	237,176
128/4,64*2	32, 128

## Find and Replace

This mode works as Find but allows the search bytes to be replaced with the ones specified in the Replace With box. The format of this box is the same as Find above, so you may specify multiple byte values or a single assembly instruction.

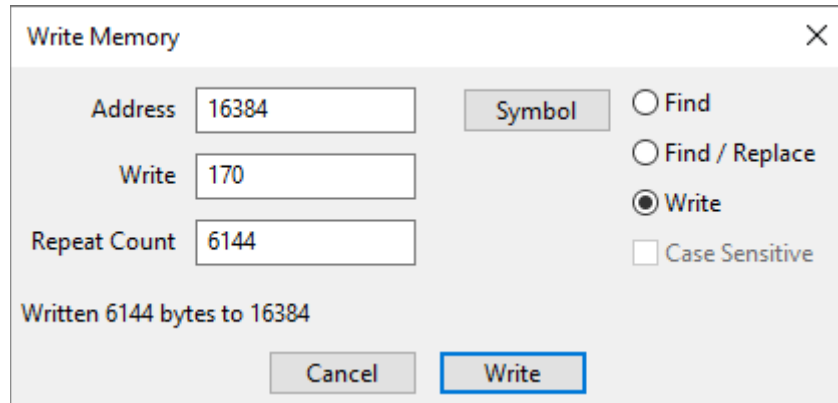


Eagle-eyed readers will note that in the screenshot above, it purports to have replaced the bytes at address \$003A, however it has not, due to that address on a ZX Spectrum being in the ROM. ROM areas are *always* read-only in Inkspector.



## Write

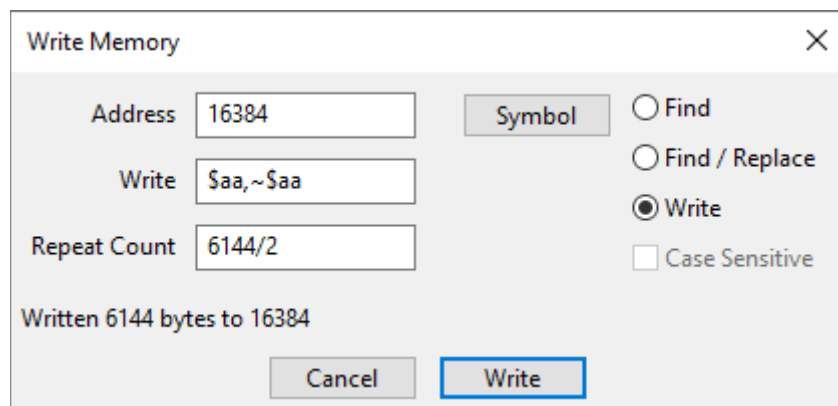
This mode doesn't actually perform a search, but writes one or more copies of the byte set described in the Write box (which is the same format as the [Find](#) box) to the address specified. Since more than one copy of the byte(s) specified in Write may be specified, this tool may be used to fill areas of memory. For example to fill the Spectrum's display memory with a striped pattern:



The screenshot shows a 'Write Memory' dialog box with the following fields and options:

- Address:** 16384
- Write:** 170
- Repeat Count:** 6144
- Symbol:** (button)
- Find:** ☐
- Find / Replace:** ☐
- Write:** ☒
- Case Sensitive:** ☐
- Status:** Written 6144 bytes to 16384
- Buttons:** Cancel, Write

Or for slightly fancier stripes (note that the repeat count has been divided by two to allow for the fact we've specified two bytes in the Write box)



The screenshot shows a 'Write Memory' dialog box with the following fields and options:

- Address:** 16384
- Write:** \$aa,~\$aa
- Repeat Count:** 6144/2
- Symbol:** (button)
- Find:** ☐
- Find / Replace:** ☐
- Write:** ☒
- Case Sensitive:** ☐
- Status:** Written 6144 bytes to 16384
- Buttons:** Cancel, Write

## Z80 Registers

This window displays the current values of the Z80's registers where they may also be modified.

The screenshot shows the 'Z80 Registers' window with a menu bar (File, View) and a title bar. The window is divided into several sections:

- Registers:** A grid of 14 registers arranged in two columns. The values are: PC: \$15E6, SP: \$FF4C, AF: \$005C, AF': \$0044, BC: \$174B, BC': \$FFFF, DE: \$0006, DE': \$5CB9, HL: \$107F, HL': \$FFFF, IX: \$FFFF, IY: \$5C3A, I: \$3F, R: \$4E. Values in red (\$15E6, \$FF4C, \$174B, \$0006, \$107F, \$FFFF, \$5CB9, \$FFFF, \$4E) indicate a change since the last refresh.
- Flags:** A row of eight checkboxes labeled S, Z, 5, H, 3, V, N, C. The checked flags are Z, H, 3, and V.
- Other:** A section containing:
  - IFF1: 1 (EI) ☒
  - IM: 1
  - INT Vector: \$0038
  - IFF2: 1
  - WZ: \$15E6
  - Affected Flags: No
  - ☐ Halted
  - ☐ Post-EI

The values in red indicate a change in value since the last refresh or debugger single-step or similar operation. The values can be modified by clicking on one and typing in an [expression](#). The flag values can be changed by clicking on their check boxes.

## System Variables

The System Variables window shows the list of variables for the machine currently running, along with their address, size in bytes, current value and brief description. The values are updated once a second. Where a variable is a pointer (e.g. CHARS) the first eight bytes of what they point to are also shown.

When open when running a 48K Spectrum, the window looks like this:

System Variables				
File View				
Name	Address	Length	Value	Description
KSTATE	\$5C00	\$08	\$FF, \$00, \$00, \$00, \$FF, \$00, \$00, \$00	Used in reading the keyboard
LAST_K	\$5C08	\$01	\$00	Stores newly pressed key
REPDEL	\$5C09	\$01	\$23	Time (in 50ths of a second in 60th
REPPER	\$5C0A	\$01	\$05	Delay (in 50ths of a second in 60th
DEFADD	\$5C0B	\$02	\$0000 -> \$F3, \$AF, \$11, \$FF, \$FF, \$C3, \$CB, \$11	Address of arguments of user def
K_DATA	\$5C0D	\$01	\$00	Stores 2nd byte of colour control
TVDATA	\$5C0E	\$02	\$00, \$00	Stores bytes of colour, AT and TAI
STRMS	\$5C10	\$26	\$0001, \$0006, \$000B, \$0001, \$0001, \$0006, \$0010, \$0000...	Addresses of channels attached t
CHARS	\$5C36	\$02	\$3C00 -> \$FF, \$FF, \$FF, \$FF, \$FF, \$FF, \$FF, \$FF	256 less than address of character
RASP	\$5C38	\$01	\$40	Length of warning buzz
PIP	\$5C39	\$01	\$00	Address of start of variables area
ERR_NR	\$5C3A	\$01	\$00	1 less than the report code. Starts
FLAGS	\$5C3B	\$01	\$00	Various flags to control the BASIC
TV_FLAG	\$5C3C	\$01	\$21	Flags associated with the televisic
ERR_SP	\$5C3D	\$02	\$FF50 -> \$7F, \$10, \$54, \$FF, \$B4, \$12, \$00, \$3E	Address of item on machine stac
LIST_SP	\$5C3F	\$02	\$0000 -> \$F3, \$AF, \$11, \$FF, \$FF, \$C3, \$CB, \$11	Address of return address from ai

It is possible to set one of the four [memory window](#) tabs to the value of a system variable by selecting the variable, then the View menu and choosing one of the four tab numbers. This also opens the memory window for you if not already open. If the variable is a pointer, the memory window is set to follow the value of the variable, by using the expression evaluator's [dereference operator](#) \*. E.g. selecting CHARS then View In Memory Tab 1, would set the address for tab 1 to “\*CHARS”.

### InkTip

To prevent the use of the deference operator when viewing a pointer variable in a memory tab, hold down CTRL as you select the menu item. “\*CHARS” would then become “CHARS” in the memory window tab.

## Watches

Watch	
File Edit View	
Watch	Value
Player1.score,a4	00000600
FRAMES,t1	\$00000000

The watch window allows an expression to be viewed in many ways, and is constantly updated while the emulated machine runs. What makes this window particularly useful is the number of ways the values can be displayed by using a modifier which may be specified by adding a comma and formatter immediately after the value being watched, as shown in the two examples above (,a4 on the first example, and t1 for the second)

Note, where [n] is accepted but not specified, 10 is assumed. So FRAMES,b10 is equivalent to FRAMES,b. Also, where no base number has been specified explicitly, the current Inkspector-wide

hexadecimal setting will be used. The examples below assume that hexadecimal numbers are being used throughout Inkspector, as per the [debugger setting](#) with ‘Display values in hexadecimal...’ being checked.

Formatter	Interpreted As	Example
a[n]	Pointer to n bytes (not digits) of BCD	score,a4 \$00000600
b[n]	Pointer to n bytes	FRAMES,b3 \$d6, \$41, \$00
w[n]	Pointer to n words	STRMS,w4 \$0001, \$0006, \$000B, \$0001
t[n]	Pointer to n, byte triplets	FRAMES,t1 \$0001F3
q[n]	Pointer to n, byte quads	STRMS,q2 \$00060001, \$0001000B
f[n]	Pointer to n, Sinclair format floating point numbers (5 bytes each)	MEMBOT,f6 10, 10, 10, 0, 0, 0
c	Show the value as single ASCII character	65,c A
e	Pointer to an ASCII string terminated by the value \$80	msg,e [“Hi!”, \$80] Hi!
h	Pointer to an ASCII string terminated by the value \$76 (Z80 HALT instruction)	msg,h [“Hi!”, \$76] Hi!
v	Pointer to an ASCII string terminated with the final character having bit 7 set	msg,v [“Hi”, “!”+\$80] Hi!
z	Pointer to an ASCII String terminated with a zero byte	msg,z [“Hi!”,0] Hi!
s[n]	Pointer to a fixed [n] length ASCII string	msg,s3 [“Hi!”] Hi!

### **InkTip**

If you change the c, e, h, v and z specifiers to upper-case, they will return an ASCII string based on the underlying machine’s character set, rather than assuming it’s ASCII.

e.g. assuming the current machine is a ZX80 and msg points to a buffer containing the bytes \$2D, \$2E,\$00 msg,Z (i.e. upper-case ‘z’) will display “HI” (the ZX80 doesn’t have an exclamation mark in its character set, BTW)

The following formatters may be used in conjunction with the ones above to adjust the number base used to display the results.

Formatter	Interpreted As	Example
x	Hexadecimal	123,x \$7B
d	Decimal (unsigned)	\$7B,d 123  STRMS,w4,d 1, 6, 11, 1
n	Signed decimal	0-1,n -1
i	Binary	123,i 0b1111011

The Edit menu contains the following items

Edit Menu Item	Function
Copy Value	Copies the value of the selected watch item to the clipboard
Add Watch	Adds a new watch item
Add Watch From Symbol	Adds a new watch item from a symbol (if available)
Edit Watch	Edits the selected watch item
Delete Watch	Deletes the selected watch item
Delete All Watches	Deletes all watch items after confirming

The View menu allows you to send the value of the currently selected watch item to one of the four memory windows, opening them if they're not already open. Additionally the Inkspector-wide hexadecimal number setting can be toggled on and off here.

## Messages Window

Inkspector likes to show information. It likes to show the details of files being loaded or saved, of files being assembled. And about lots of other things generally. To see this additional information and messages, you must have the Messages window open (View → Messages or press Alt-M). In fact, I find this window so useful, I added an option to the [Start-up and Shut Down](#) page specifically to ensure it's opened when Inkspector runs.

Just one example of this window being useful is if you were to encounter a snapshot file that Inkspector wasn't happy loading. Although an error message containing the basic reason for the failure is shown as a matter of course, having the messages window open before you attempted the load would show in detail what the issue was.

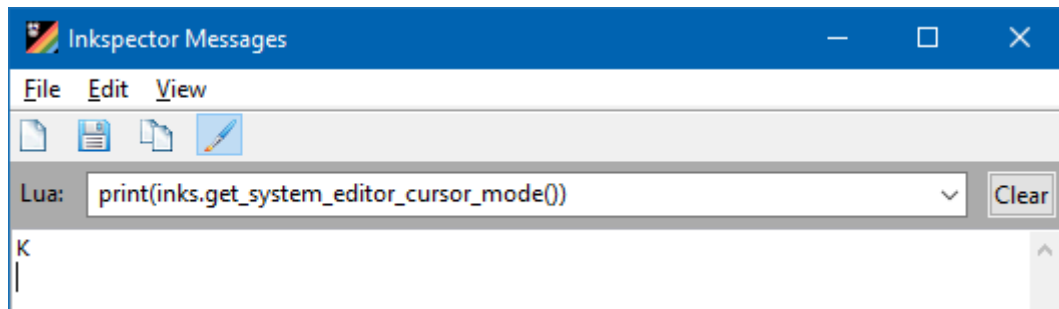
Because the amount of information shown can sometimes be overwhelming, errors are shown in red to make it easier to identify them. Blue text is used for warnings, and green when Inkspector is feeling smug and wants to share some particular success with you.

When Inkspector generates a message, it always includes a time-stamp. However, I find – for the messages window anyway – that this extra information doesn’t normally add anything useful, and so it’s possible to show or hide the time stamp part of the message using the Message window’s View → Show Timestamps menu item. As we will find out [later](#), messages can also be written to disk and I do find the time stamps useful there, which is the reason they continue to be generated.

The four letter prefix on each message, e.g. “[MACH]” indicates which part of the Inkspector code produced it. As with timestamps, they can be hidden by selecting View → Show Headers.

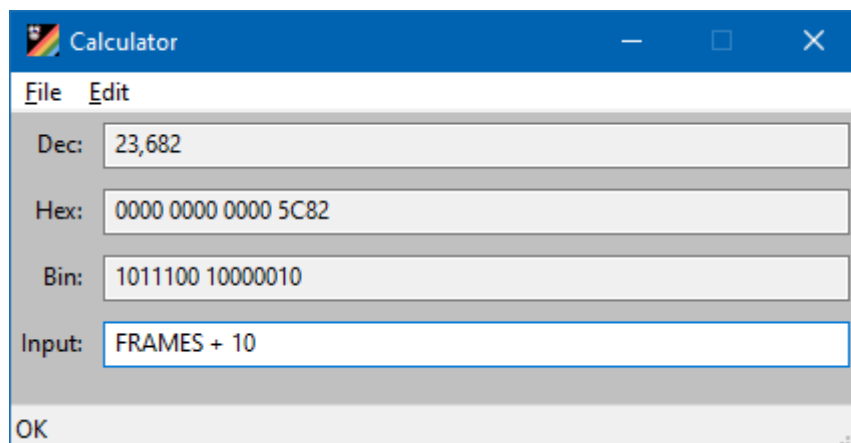
One note about the Copy functionality on this window: If Copy is performed with no text selected, all text is copied to the clipboard, otherwise just the selected portion. File → Save As saves all the text to a file regardless of what’s currently selected.

As well as displaying messages, this window allows you to enter a single line Lua command (although as with standard Lua, you can specify multiple commands separated by a semi-colon). This can be useful when just wanting to enter one-off commands without having to sacrifice a snippet. And since often you want to display some sort of result having run the command, having this ability on the message window makes it particularly convenient.



## Calculator

The calculator is a simple expression evaluator that displays the result in multiple number bases at once. The results are updated automatically as the input is typed in. As usual, symbols and the Z80 registers [may be referenced](#).



## Assembler

Inkspector contains a full-featured Z80 macro assembler that may be accessed in several ways. The most straightforward method to assemble a file is to load one with a .s or .asm extension into the

GUI or CLI as a regular supported file, where it will be assembled immediately. Individual Z80 instructions may be assembled directly in the debugger and memory windows by right-clicking on the address you wish to assemble to and selecting “Poke” from the menu that appears then clicking on the “Assemble” radio button on it. You can also enter Z80 instructions when using the [Find Memory](#) tool as the item to search for. The assembler is also available via [scripting](#) and via the Tools → Assemble Clipboard Contents menu item.

## Syntax

The assembler expects source files to be formatted with labels specified on the left-most column (i.e. no whitespaces in front of them), optionally followed by a [directive](#) (in uppercase or lowercase) or Z80 instruction. Comments begin with a semicolon, indicating the rest of the line will be ignored.

Labels may consist of alphanumeric characters, underscores and full stops. Local labels start with a full stop.

The assembler uses standard Zilog Z80 mnemonics (<http://www.zilog.com/docs/z80/um0080.pdf>) but also includes all known undocumented instructions, including the ability to use IX and IY’s component 8-bit registers IXL, IXH, IYL and IYH.

The special symbol \$ represents the current address being assembled to.

The following lines demonstrates some of the above:

```
                org 32768                ; This is a comment
Start:         ld hl,$4000                ; Define the label “Start” at address 32768

.loop:         inc    a                    ; Label local to Start
                jr     nz,.loop            ; jumps to .local above

Another:
.loop:         inc    a
                jr     nz,.loop            ; jumps to .local above, not the one immediately below
Start
```

## Directives

=

An alias for [DEFL](#)

```
LabelValue = 1234
```

### ALIGN n

n is a value between 1 and 32768.

The current address is aligned on an ‘n’ byte boundary.

```
ORG    $8001
ALIGN  16                ; Align the current address to a 16-byte boundary
ASSERT $==$8000+16      ; The current address is now $8010
```

## ASSERT [expr]

Halts the assembly if expression *expr* does not evaluate to 1. If *expr* is not specified, 0 is assumed, which will always halt assembly.

See the example for [ALIGN](#).

## ASSERTIFDEF identifier

Halts the assembly if identifier exists.

```
ASSERTIFDEF ThisShouldNotExist
```

## ASSERTIFNDEF identifier

Halts the assembly if the identifier does not exist.

```
ASSERTIFNDEF ThisShouldExist
```

## ASSERTSTR S1,S2

Halts the assembly if the string S1 is different to S2. The strings may be quoted strings or string identifiers.

```
DEFINE      A      "A"
ASSERTSTR   A,      "A"
DEFINE      A2     A
ASSERTSTR   A2,     A
```

## BANK bank-num

*bank-num* is a value between 0 and 7.

Sets the current RAM page to *bank-num* and the current address to the start of the bank to 49152 (\$C000). Currently this directive is only supported for [targets](#) of ZXSPPECTRUM128 and later.

```
TARGET ZXSPPECTRUM128
BANK 1
ASSERT $ == $C000 ; Verify the current address is now $C000
```

## BREAK [type] [,parameters...]

BREAK instructs Inkspector's debugger to create a breakpoint for the generated code and/or any generated snapshot. For example, in its simplest form with no additional parameters, the following use instructs the debugger to break at the specified line:

```
go:    ORG $8000
        BREAK                ; The debugger will break here (alias for BREAK HERE)
        ret
        END go
```

Will cause debugger to break when the instruction at \$8000 (i.e. immediately after the BREAK directive) is executed.

### InkTip

BREAK works without injecting anything into the assembled code, so it's safe to use for 'production' code too. For example, it's safe to leave in BREAK WRITE directives that check for code writing into areas they shouldn't be, and still ship the resulting assembled code.



<b>BREAK type</b>	<b>Description</b>
(non specified)	Same as BREAK HERE
HERE	Break at the current address (\$)
BRANCH	Break at the current (or specified) address if the instruction there is a conditional and about to take the branch. e.g. a BREAK BRANCH immediately before a 'ret z' will break when the zero flag is set.
NO_BRANCH	As BRANCH, but breaks when the branch will not be taken.
READ	Break when the Z80 reads the current (or specified) memory address.
WRITE	Break when the Z80 writes to the current (or specified) memory address.
READ_WRITE	Break when the Z80 reads or writes the current (or specified) memory address.
PORT_READ	Break when the Z80 reads the specified port address
PORT_WRITE	Break when the Z80 writes to the specified port address
PORT_READ_WRITE	Break when the Z80 reads or writes the specified port address.
CONDITION	Break when the condition is true. e.g. BREAK CONDITION hl<1000 to break when the value of the Z80 register HL goes below 1000.
DI_HALT	Break when a HALT instruction is executed when maskable interrupts are disabled
INT_TAKEN	Break when a maskable interrupt is taken
NMI_TAKEN	Break when a NMI is taken
INT_RETRIGGERED	Break when a retrigged interrupt occurs

After specifying the BREAK type, one or more of the following BREAK parameters may be used, separated by commas.

<b>BREAK parameter</b>	<b>Effect</b>
CONDITION	Breaks when the CONDITION specified is true (i.e. not equal to 0) in addition to the requirements of the BREAK type. e.g. to break at the line where BREAK is placed when the Spectrum's border is green: BREAK HERE,CONDITION @border==4
COMMENT	Sets an optional breakpoint comment that's displayed and logged when the breakpoint is hit. If a COMMENT is not set, the BREAK command and its parameters is used as the comment instead.
ONESHOT	Specifies the breakpoint is one-shot, i.e. it deletes itself once hit.
SNIPPET	Specifies a snippet to run in the range 1 to 10 when the breakpoint hits.
LENGTH	When used with memory read/write or port read/write breakpoints, specifies the length of the memory area to monitor. The default is a length of 1. e.g. to break when the memory between 40000 and 49999 is written to:

	BREAK WRITE,ADDRESS 40000,LENGTH 10000
VALUE	When used with memory read/write or port read/write breakpoints, specifies the value that must be read or written before the breakpoint will fire. So to add to the above example, but only fire when the value 123 is written between addresses 40000 and 49999: BREAK WRITE,ADDRESS 40000,LENGTH 10000,VALUE 123
ADDRESS	Sets the address for the BREAK statement
RAM	Sets the RAM page applicable to ADDRESS above (mutually exclusive with ROM)
ROM	Sets the ROM page applicable to ADDRESS above (mutually exclusive with RAM)
ADDRESS_MASK	Sets the address mask for this breakpoint. See <a href="#">Add Breakpoint</a> for details on how this value is used

In addition, you can use all the options that are available in the GUI's Edit Breakpoint dialog window. For example using the conditional breakpoints in conjunction with the pseudo variables (ones starting with @), to make the debugger break if the Spectrum's border turns green:

```

ORG $8000
go:  BREAK CONDITION @border==4
      ld    a,4
      out   ($fe),a
      ; Debugger will break here
      ret
      END go

```

If you want to catch writes to memory, you can do that too:

```

ORG    $8000
; 1000 = number of bytes to monitor so, between "go" and "go"+999 inclusive.
BREAK WRITE,ADDRESS go, LENGTH 1000
go:    ld    hl,0
      ld    de,go-200
      ld    bc,1000
      ; The debugger will break at the point the memory at $8000 is written to
      ldir
      END    go

```

If you wish to break when a specific value is written (or read) the VALUE option allows you specify it:

```

ORG    $8000
; Stop when 4 is written to livesLeft
BREAK WRITE,ADDRESS livesLeft, VALUE 4
go:    ld    hl,livesLeft
      dec   (hl)
      jr    nz,go
      ret
livesLeft db    8
      END    go

```

Here's a couple of examples showing the use of BREAK BRANCH

```
ld          hl,0
BREAK BRANCH ; Will break when the zero flag is set
ret         z

ld          hl,0
BREAK NO_BRANCH ; Will break when branch is not taken (i.e. when B == 1)
djnz       lp
```

All breakpoints requested by BREAK directives are created in the machine that receives the assembled code so they take effect in the debugger immediately, as you would expect.

For cases where BREAK is used and at least one of the assembler's output is a snapshot file, a snapshot auxiliary file is created in the same directory as the snapshot file. This is to allow the breakpoints to persist after Inkspector is closed down, and automatically loaded and applied the next time the snapshot is loaded.

To be able to do this, auxiliary snapshot files have the same name as the snapshot, but with .aux.xml added. For example if the assembler generated c:\temp\myprog.szx, it would also create c:\temp\myprog.szx.aux.xml. As well as the list of breakpoints, it may also in future contain additional information to expand the abilities of existing snapshot formats such as adding support for hardware not supported by their original specifications.

If the BREAK directives are removed or commented out and the code assembled again, the snapshot auxiliary files will be deleted so that old breakpoints are no longer applied when Inkspector loads the snapshot.

Since the BREAK breakpoints are applied using auxiliary files, the assembled output, and therefore the snapshot file itself, is not modified by the use of BREAK and so may be used to distribute a program or game without issue (i.e. no opcodes or such are injected into the assembled code by the BREAK directive).

NB for the auxiliary files to be written, deleted or loaded with a snapshot, auxiliary snapshot file support must first be enabled in Inkspector. This is done on the GUI's Assembler options page by enabling "Use snapshot auxiliary files".

For full details on the BREAK directive and auxiliary files, see the documentation.

## DEFINE name value

Defines an identifier named *name* with value *value*. *value* is optional and may be a numeric or string value. Once created, *name* may not be re-defined or assigned another value. Use [DEFL](#) if you wish to create a modifiable identifier.

```
DEFINE IntM2VectorTableAddr      $8200
DEFINE Author                    "Mark Incley"
DEFINE A_Big_Value_That_Would_Be_Too_Large_For_A_Label 1234567890
DEFINE I_Have_No_Value_But_I_Am_Still_Valid
```

## DEFL name value

Creates an identifier named *name* with the value *value*. Unlike identifiers created with [DEFINE](#), these may be re-assigned different values.

```

DEFL          Count  1
DEFL          Count  Count + 1      ; Would error if created with DEFINE
ASSERT Count==2

```

## DEFNM msg

Defines a message using the character set for the native machine (set by [TARGET](#)). For Spectrum and Jupiter ACE machines, which use (almost) ASCII compliant character sets, the results are the same as if defining a message using DB msg. For the ZX80 and ZX81 machines, the bytes produced are for their native character sets which do not resemble ASCII and also have a greatly reduced range of characters, for example, no lower-case, no exclamation marks, etc.

Where characters cannot be substituted for the native character set (e.g. upper case for lower case) a question mark is used.

```

TARGET ZX80
DEFNM "Hi!" ;      Produces $2D,$2E,$0F (native: HI?)

TARGET ZX81
DEFNM "Hi!" ;      Produces $2D,$2E,$0F (native: HI?)

TARGET ZXSPECTRUM
DEFNM "Hi!" ;      Produces $48,$69,$21 (native: Hi!)

TARGET ACE
DEFNM "Hi!" ;      Produces $48,$69,$21 (native: Hi!)

```

## DEVICE target-device

This is an alias for [TARGET](#) for compatibility with SjAsmPlus.

## DISPLAY msg

Displays a message. When assembling under the command line version of Inkspector, the message is displayed on the console window. For the GUI, it is displayed on the message window. Multiple items may be displayed by separating them with a comma.

```

ORG 32768
DISPLAY "The current address is ",$      ; Displays "The current address is 32768"

```

## ELSE

Reverse the condition from the IF-family directive that precedes it and start processing or ignoring the subsequent lines as appropriate until the next [ENDIF](#). See [IF](#) for an example.

## END [start-address]

Indicate the end of the assembly, optionally setting the start address of the program. Any lines following END are ignored.

```

END ProgStart      ; We want to the program to start execution at ProgStart
RHUBARB            ; The assembler won't see this "directive"

```

## ENDIF

Marks the end of the most recent IF-family directive. See [IF](#) for an example.

## ENDM

Marks the end of a macro definition. See [MACRO](#).

## ENDPOKTRAINER

Ends the current trainer definition. This directive is also called implicitly by the [ORG](#), [BANK](#), [END](#) and [POKTRAINER](#) directives, and when the current assembly ends.

## ENDR

Ends a [REPT](#) repeat block.

## ENDS

Ends a [STRUCT](#) definition.

## EQU address[,bank]

Creates a label with an address other than \$, optionally specifying a RAM bank.

```
MyLabel      EQU    $1234
MyLabelBank  EQU    $1234,7      ; Address $1234 in bank 7
```

## EXECUTE execution-mode

*execution-mode* is AUTO, IMMEDIATE or TAPE

Determines how the emulator should execute the assembled program.

Execution Mode	Effect
IMMEDIATE	The assembled program in memory is executed immediately, even if a tape image has been generated.
TAPE	The assembled program is executed by loading the generated tape image. If there is no tape image, a warning is shown and an IMMEDIATE execution is attempted.
AUTO	If a tape image has been generated, it behaves as TAPE, otherwise as IMMEDIATE. This is the default execution mode.

Generally you would want to use EXECUTE IMMEDIATE when the generated program is self-contained. i.e. does not rely on the system's OS having booted up, initialised and ready to be called upon. If you are writing a program that does require the OS to be active, generate a tape image which allows the emulator to boot up the OS then load the generated tape image automatically.

## IF expr

If *expr* evaluates to a non-zero value, assemble the following lines up until the next [ELSE](#) or [ENDIF](#), otherwise ignore them until that point.

```
IF Infinitelives==0
ld    hl,lives
```

```

dec    (hl)
ELSE
DISPLAY "Infinite lives are active!"
ENDIF

```

## IFDEF identifier

If *identifier* is defined, assemble the following lines up until the next [ELSE](#) or [ENDIF](#), otherwise ignore them until that point.

## IFLABEL label

If *label* is defined, assemble the following lines up until the next [ELSE](#) or [ENDIF](#) otherwise ignore them until that point.

## IFNDEF identifier

If *identifier* is not defined, assemble the following lines up until the next [ELSE](#) or [ENDIF](#) otherwise ignore them until that point.

## IFNLABEL label

If *label* is not defined, assemble the following lines up until the next [ELSE](#) or [ENDIF](#) otherwise ignore them until that point.

## IMPORT item

Imports *item* into the assembler. Currently the only *item* available is SYSVARS. IMPORT SYSVARS creates a set of labels from the system variable names for the current machine (specified with [TARGET](#)).

```

TARGET ZXSPCTRUM48
IMPORT SYSVARS           ; This brings in about 70 system variable names
ASSERT KSTATE==$5C00
ASSERT FRAMES==$5C78
ASSERT P_RAMT==$5CB4

```

Or with a Jupiter ACE

```

TARGET ACE
IMPORT SYSVARS
ASSERTIFDEF KSTATE      ; Spectrum vars shouldn't be here!
ASSERT DICT==$3C39

```

## INCBIN filename

Adds the contents of the file *filename* at the current address.

## INCHEX filename [address]

Includes the decoded contents of the [Intel hex](#) format file *filename* at the address specified in the file (or *address* if specified as a second parameter)

## INCLUDE filename

Assembles the contents of *filename* at the place this directive is used. Nested includes (INCLUDE directives within files that are themselves INCLUDED) are supported up to 30 levels deep.

## LABELSLIST

This is an alias for [SAVESYMBOLS](#) for SjAsmPlus compatibility.

## MACRO name [parameters...]

Defines a macro called *name* with one or more optional parameters. Once the macro has been defined, it may be “played back”, where it is then expanded into the current assembly address. The macro definition must end with an [ENDM](#) directive.

```
MACRO SETBORDER border
ld    a,border
out   ($fe),a
ENDM
```

This simple macro sets the border colour on a ZX Spectrum. Once defined, it can be used thus:

```
SETBORDER 4 ; Set the border to green
```

and will produce the following code:

```
ld    a,4
out   ($fe),a
```

Macros can also expand other macros:

```
MACRO BLACKBORDER
SETBORDER 0
ENDM
```

to produce

```
ld    a,0
out   ($fe),a
```

This macro calculates the Spectrum screen address for a pixel at x (left 0-255) and y (top 0-191) and places it in the designated register pair:

```
MACRO CALCSCR reg,x,y
ld    reg,$4000+((x)/8)+(256*((y)&7))+($20*(((y)&$3f)>>3))+(((y)>>6)*$800)
ENDM
```

With

```
CALCSCR de,128,100
```

producing

```
ld    de,$4C90
```

Local labels may be used inside a macro and will not cause redefined label errors when the macro is used more than once:

```

MACRO LOCALTEST
or      a
jr      nz, .notz

cpl

.notz: ENDM

LOCALTEST
LOCALTEST ; Just to prove MACRO local labels don't produce duplicate label
errors!

```

## MEMCOPYMODE mode

Specifies the method used to copy the generated assembly to the target machine once assembly has completed successfully. *mode* must be either NEW, SPARSE or EXTENT.

NEW causes a new instance of the existing machine to be created, with its RAM cleared to zero, and then has the generated assembly copied to the machine's memory as per the SPARSE mode.

SPARSE copies the generated assembly over the existing machine's memory, leaving areas of memory not covered by the assembly untouched. This allows, for example, two sections of code with their own ORG addresses to be copied over without disturbing memory areas in-between.

EXTENT copies the generated assembly over the existing machine's memory but unlike SPARSE, all memory between the minimum and maximum assembled addresses is also copied over regardless of whether the addresses in-between contained assembled output.

Assembler Invoked From	Default MEMCOPYMODE
.s, .asm file loader	NEW
Script (calling <a href="#">inks.assemble</a> )	SPARSE
GUI's Assemble from Clipboard	SPARSE

To visually demonstrate the different modes, select a Spectrum 48k and copy and paste the following into a spare [snippet](#):

```

inks.assemble(
[[
    ;MEMCOPYMODE NEW
    ;MEMCOPYMODE SPARSE
    ;MEMCOPYMODE EXTENT

    MACRO LINE ADDR
    ORG ADDR
    REPT 32
    db      ($&7) << 3
    ENDR
    ENDM

    LINE $5820
    LINE $59E0
]]

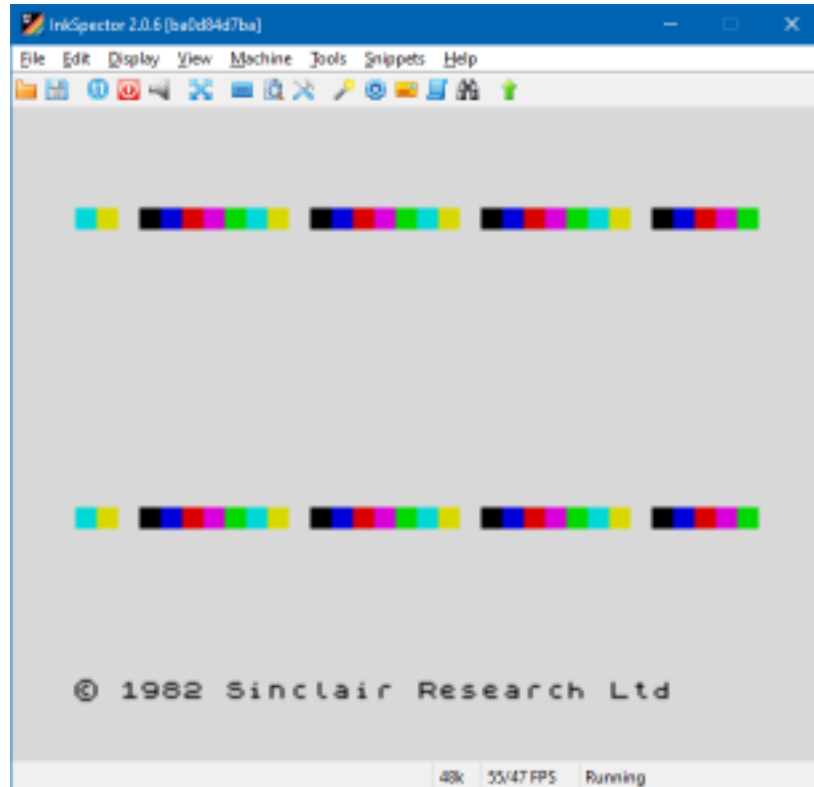
```

Now run the snippet three times, each time enabling just one of the ;MEMCOPYMODE lines by removing the leading semicolon.

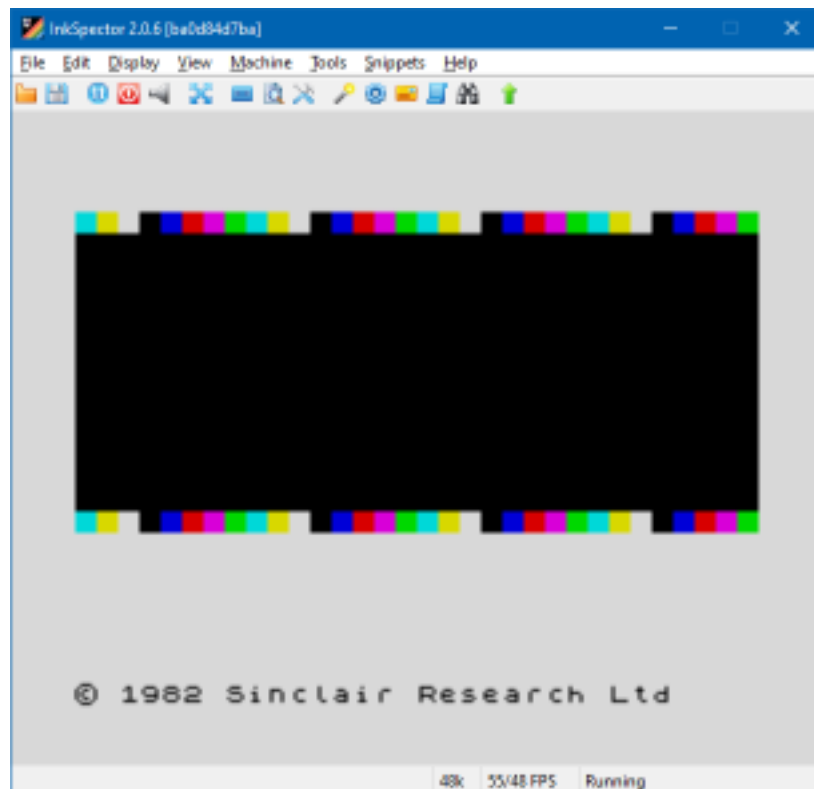


With NEW selected, the machine will reboot and you will end up with the usual welcoming copyright message. The two sets of coloured lines won't be visible as they've been overwritten by the Spectrum ROM's RAM testing as it boots up.

With SPARSE, both blocks of assembled code are copied over separately into the machine without disturbing any other memory:



With EXTENT, the entire area between the two blocks of code are copied over (i.e. including the unassembled zero bytes between the two blocks of code)



## ORG address

Set the current assembly address to *address*.

## POKTRAINER trainer-name

Creates a POK trainer named *trainer-name*. [SAVEPOK](#) must be used before the first POKTRAINER. The trainer continues to be produced until another POKTRAINER is specified, [ENDPOKTRAINER](#) or [BANK](#) is used, or until assembly is completed. See [SAVEPOK](#) for an example.

## REPT repeat-count

Repeats the block of code within it *repeat-count* times. Repeat counts of 0 are valid, which essentially ignores the code within the REPT block. The block must end with [ENDR](#).

ShowBackBuffer:

```
1d    h1,0ffffh
1d    de,57ffh
1d    bc,1000h
```

```
.loop:    REPT    16
          ldd
          ENDR
          jp     pe,.loop
          ret
```

## SAVEBIN [filename]

Saves the raw assembled bytes to a file. If *filename* is not supplied, the name of the main source file with an extension of “.bin” is used.

## SAVEHEX [filename]

Saves the assembled bytes to an [Intel Hex](#) format file. If *filename* is not supplied, the name of the main source file with an extension of “.hex” is used.

## SAVELISTING [filename]

Generates a listing file. If *filename* is not supplied, the name of the main source file with an extension of “.lst” is used.

The listing file contains every line of source assembled, along with bytes generated. The first column shows the line number within the current source file. If the current file has been [INCLUDE](#)d the line number is followed with a plus sign.

The second column shows the current assembly address. The third column shows either the bytes generated from Z80 instruction on the line or a tilde if the line is declaring a [MACRO](#) or [STRUCT](#).

This is followed by the original source file itself.

If the UNREFS directive has been used, the listing file will include a report of any labels, identifiers, macros and structs that were declared but not referenced.

## SAVEPOK [filename]

Generates a Spectrum Games Database format POK file. If *filename* is not supplied, the name of the main source file with an extension of “.pok” is used. After using SAVEPOK, trainers are added by using [POKTRAINER](#).

For example:

```
SAVEPOK      "test.pok"

POKTRAINER   "Trainer One"
ORG          0x8052
jp          0xB701
ORG          0xB701      ; Note multiple ORGs is fine in a trainer
ret

POKTRAINER   "Trainer Two"
ORG          0xC000
ld          a,ix
ret
```

will product a file called test.pok that looks like this:

```
NTrainer One
M 8 32850 195 0
M 8 32851 1 0
M 8 32852 183 0
Z 8 46849 201 0
NTrainer Two
```

```

M 8 49152 221 0
M 8 49153 125 0
Z 8 49154 201 0
Y

```

## SAVESNA

This is an alias for [SAVESNAP](#) for SjAsmPlus compatibility.

## SAVESNAP filename [,start-address]

Saves the generated assembly as a snapshot file with the given *filename*. If *start-address* is specified it is used to set the start address in the resulting snapshot file, otherwise the address specified by [END](#) is used.

## SAVESYMBOLS filename

Saves a list of symbols and identifiers to file *filename*. The format of each line is the same as SjAsmPlus, namely:

```
[pp]:value name
```

Where *name* is a label (i.e. not an identifier), *pp* is the RAM page number (0-7) and *value* is the hexadecimal address within the page (0000-3FFF). Where *name* is an identifier, *pp* is omitted and the value is shown unmodified.

Example extract showing the values of three identifiers followed by two labels:

```

:0002 EnemyType_Helicopter
:0004 CoordinatesPerPixel
:CE00 InterruptVectorTable
02:24DC NewGame
00:0311 BaddiesBullets

```

## SAVETAP filename[,start-address]

For compatibility with SjAsmPlus, this directive creates a tape image called filename (see [Tape Loader Generators](#) for the supported tape image types). If start-address is specified it overrides any program start address specified with [END](#).

## SAVETAPE filename[,loading-screen-filename]

Similar to [SAVETAP](#), this directive creates a tape image called filename (see [Tape Loader Generators](#) for the supported tape image types). If *loading-screen-filename* is specified, it is used as a loading screen in the generated tape image (ZX Spectrums only).

NB with SAVETAPE the program start address must be set by the [END](#) directive.

## SETREGISTER register, value [,register, value...]

Sets the values of one or more Z80 registers prior to saving any snapshots specified with [SAVESNAP](#).

```

SETREGISTER    sp,$5E22
SAVESNAP        "test.sna"    ; SP will be set to $5E22 when the snapshot is written

```

## STRUCT name [,initial-offset | base-struct]

Creates a structure called *name*. Every line between the STRUCT and the ENDS directive will be added to the structure. Every line containing an instruction (e.g. “DB”) must have a label declared. Local labels are disallowed within a STRUCT block.

```
                STRUCT Sprite
x               DB    0           ; X coor
y               DB    0           ; Y coor
gfx            DW    0           ; Pointer to graphic
                ENDS

; Create our STRUCT instance
                ORG    $8000
mySprite       Sprite           ; Produces $00,$00,$0000 (4 bytes total)
```

A STRUCT may be referenced in the following ways

```
; Addressing the STRUCT directly gives you its total size
                ASSERT Sprite == 4
                ASSERT $ == $8000 + Sprite
                ld     de,Sprite

; Addressing a STRUCT's member gives you its offset
                ASSERT Sprite.x == 0
                ASSERT Sprite.y == 1
                ASSERT Sprite.gfx == 2
                ld     b,(ix+Sprite.y)

; Addressing a STRUCT variable's member gives you its address
                ASSERT mySprite.gfx == $8002
                ld     hl,(mySprite.gfx)
```

It's also possible to have a structure's first entry to start at a value other than zero by specifying an initial offset after the STRUCT name.

```
                STRUCT Sprite, 4
x               DB    0           ; X coor
y               DB    0           ; Y coor
gfx            DW    0           ; Pointer to graphic
                ENDS

                ASSERT Sprite.x == 4
                ASSERT Sprite.y == 5
                ASSERT Sprite.gfx == 6
```

Wait! There's more! You can even base a STRUCT on an existing one. Taking our original Sprite STRUCT with an initial offset of 0:

```
is_firing      STRUCT Baddie,Sprite
                db     0
                ; etc.
                ENDS

                ASSERT Baddie.x == 0
                ASSERT Baddie.y == 1
                ASSERT Baddie.gfx == 2
                ASSERT Baddie.is_firing == 4
```

This can be useful when wanting to re-use the same structure in your code for different objects, e.g.

```
STRUCT Player,Sprite
db    numLives;
; etc.
ENDS

ld    ix,Player
call  DrawSprite

ld    ix,Baddie
call  DrawSprite
ret

DrawSprite: ld    c,(ix+Sprite.x)
            ld    b,(ix+Sprite.y)
            call  CalcBCToScreen
            ; etc.
```

Finally, if, for some reason, you want to base a STRUCT on an existing one but don't want to inherit its members, specify the initial offset as a plus sign followed by the existing STRUCT's name:

```
is_firing STRUCT AltBaddie, +Sprite
db    0
; etc.
ENDS

; NB no x, y, or gfx members inherited from Sprite
ASSERT AltBaddie.is_firing == 4
```

## TAPEINFO category:text

Adds information to the [generated tape images](#) (currently for .tZX and .pZX images only). The category must be one of the ones as set out in the TZX specification (also used by the .pZX format). i.e. Title, Publisher, Author, Year, Language, Type, Price, Protection, Origin and Comment.

```
English    TAPEINFO    "Title: Example of Inkspector's fab tape generator"
           TAPEINFO    "Publisher: Andy Sturmer"
           TAPEINFO    "Author: Roger Joseph Manning Jr."
           TAPEINFO    "Year: 1989"
           ;TAPEINFO    "Language: English"          ; NB not necessary if language is
           TAPEINFO    "Type: Demo"
           TAPEINFO    "Price: 0.00"
           TAPEINFO    "Protection: None"
           TAPEINFO    "Origin: Original"
           TAPEINFO    "Comment: Bellybutton"
           TAPEINFO    "Comment: Spilt Milk"
           TAPEINFO    "Comment: And no more"
           SAVETAPE    "my-lovely-tape.tzx"          ; Create tape with the info above
```

## TAPELOADEREXEC command

Specifies which BASIC command the [generated loader](#) will use to execute the assembled program once loaded, when generating a tape loader for the ZX80, ZX81 or the ZX Spectrum. The default command is PRINT.

command	Example Generated BASIC
PRINT	PRINT USR 24576
RAND	RANDOMIZE USR 24576
LET	LET R=USR 24576

```
TAPELOADEREXEC RAND
SAVETAPE      "prog-will-be-started-with-a-randomize-command.tzx"
```

## TAPEPROGNAME name

When used with SAVETAPE, it overrides the default filename used for the first file in the generated tape image.

```
SAVETAPE      "my_tape.tap"
TAPEPROGNAME  "ace prog"
```

## TARGET machine

Specifies which machine the program is targetted for. **Machine** may be one of the following:

Target machine	Value for machine
16k ZX Spectrum	16k or ZXSPECTRUM16
48k ZX Spectrum	48k or ZXSPECTRUM48
128k ZX Spectrum	128k or ZXSPECTRUM128
Spectrum +2	plus2 or ZXSPECTRUMPLUS2
Spectrum +2A	plus2a or ZXSPECTRUMPLUS2A
Spectrum +3	Plus3 or ZXSPECTRUMPLUS3
Sinclair ZX80	zx80
Sinclair ZX81	zx81
Jupiter ACE	ace or JUPITERACE

If the specified machine is different to the one currently running, the specified one is selected. If no TARGET directive is used, the machine active when the assembler is invoked is used.

Instructing the assembler which machine you're targetting enables or disables [bank paging](#) as appropriate. For the ZX80, ZX81 and Jupiter ACE it also sets the ORG address assuming that tape images are going to be made (the ORG addresses will place the assembled code in the REM line hosting the code).

## TRACE [OFF | ON]

Enables (if *ON* or no parameter is specified) or disables the assembler's trace (diagnostics) level. Turning this on produces a lot of information when assembling a file and isn't recommended unless diagnosing some assembler issue. The default trace setting is off.

## UNREFS

Display a list of unreferenced objects at the end of a successful assembly.

If [SAVELISTING](#) has been used, the same list of unreferenced objects is appended to the listing file.

The list of unreferenced objects includes labels, identifiers, [STRUCTs](#) and [MACROs](#). Unreferenced labels imported by [IMPORT SYSVARS](#) are not shown.

## VERBOSE [OFF | ON]

Enables (if *ON* or no parameter is specified) or disables the assembler's verbose mode. Turning this on produces more information when files are assembled. The default is "on" if "Show additional messages that may be useful..." is enabled in the GUI's Advanced options page, otherwise off.

## Tape Loader Generators

The assembler has the ability to create a tape image containing a loader for the generated assembly for most of the machines emulated by Inkspector. The current exception is the Jupiter ACE.

The [TAPELOADEREXEC](#) directive works for all machines shown below.

### ZX80

The ZX80 tape loader generator can produce .o or .80 tape image files from the assembler output. Assembling the example `s\example_zx80.s` supplied with Inkspector produces a tape image containing the following BASIC program with the assembler output stored in the REM statement at line 40.

```
10 REM MADE BY INKSPECTOR 2.0.6 (DCB7894348)
20 PRINT USR(16488)
30 STOP
40 REM
```

In an attempt to stop the ZX80 ROM crashing when listing the REM at line 40 that could potentially contain characters that could upset the delicate beast, the first two characters after REM and immediately before the program starts, fool the ROM into thinking it's the end of the line so it doesn't attempt to continue to draw the remaining characters, so you don't see the usual expected selection of seemingly random characters associated with putting machine code in a REM statement.

### ZX81

The ZX81 tape loader generator can produce .81, .p or .p81 tape image files from the assembler output. Assembling the example `s\example_zx81.s` supplied with Inkspector produces a



tape image containing the following BASIC program with the assembler output stored in the REM statement at line 20.

```
10 REM MADE BY INKSPECTOR 2.0.6 (DCB7894348)
20 REM 5sRND/ SAVE HELLO FROM INKSPECTOR.\067
30 PRINT USR VAL "16557"
```

## ZX Spectrum 16k, 48k

The ZX Spectrum tape loader generator can produce .tap, .ttx and .pzx tape image files from the assembler output. Assembling the example snapshots\s\example.s supplied with Inkspector produces a tape image containing the following BASIC program to load the assembled output.

```
10 CLEAR 32767
20 PRINT "Tape created by Inkspector"
30 PRINT "v2.0.6 [dcb7894348]"
40 PRINT
50 PRINT "Loading 1 files..."
60 LOAD "main mem"CODE
70 PRINT USR 32886
```

## ZX Spectrum 128K and later

The 128k ZX Spectrum tape loader generator can produce .tap, .ttx and .pzx tape image files from the assembler output and supports the assembler's [BANK](#) directive to handle memory pages. Assembling the example snapshots\s\example\_128k.s supplied with Inkspector produces a tape image containing the following BASIC program to load the assembled output.

```
10 CLEAR 25999
20 PRINT "Tape created by Inkspector"
30 PRINT "v2.0.6 [dcb7894348]"
40 PRINT
50 PRINT "Loading 3 files..."
60 LOAD "128k_sup"CODE
70 IF NOT USR 26001 THEN PRINT FLASH 1;"128K Machine Required": STOP
80 POKE 26000,1: RANDOMIZE USR 26003
90 LOAD "page 1"CODE
100 LOAD "main mem"CODE
110 PRINT USR 32768
```

## Scripting

Inkspector uses the Lua 5.4 scripting language (<https://www.lua.org>), containing out-of-the-box Lua functionality with extensions added to provide access to Inkspector's features, from basic load and save operations to being able to generate disassemblies, manipulate the currently emulated machine and [a lot more](#).

To make scripting a little more accessible and easier to use, Inkspector has the concept of snippets, which are small Lua scripts intended to run simple tasks, leaving stand-alone script files for long-running tasks or anything a bit more adventurous. There are 10 pre-defined snippets that may be edited within Inkspector as you wish, and you can run them in the GUI by pressing their shortcut keys. Alt-1 to run snippet #1, Alt-2 snippet #2, etc., up to Alt-0 to run snippet #10.

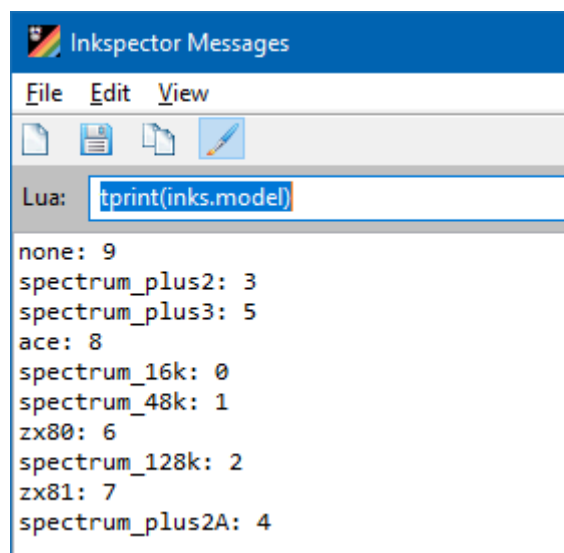
Script files may be run in the following ways:

- By loading a file with a .lua extension. Inkspector assumes such files are Inkspector script files, which are UTF-8 text files containing Lua script.

Snippets may be executed in the following ways:

- Directly from within the GUI's snippet editor
- From the GUI's Snippets menu (shortcut keys Alt-1, Alt-2, etc.)
- Selecting one to be run on startup of the Inkspector GUI by selecting one from Options → Startup and Shutdown, or pressing the Start Up button on the Snippet Editor.
- Selecting one to be run when a debugger breakpoint is hit
- Using the CLI's --snippet option (e.g. --snippet 1 to run Snippet #1)

You can also run single-line Lua commands from the [messages window](#) by typing them into its Lua bar (View → Lua Bar or press the fourth toolbar button along as shown below) and pressing ENTER. I find this useful when just wanting to see the output of a print or tprint command as the output appears immediately below in the message area. For example, showing all of the values in the inks.model table:



Lua was selected for Inkspector because despite being powerful, it's easy to learn and use. If you are not already familiar with Lua and would like to use Inkspector's scripting support, I'd recommend starting with this web page: <https://www.lua.org/start.html>. Or you could just take the example scripts and snippets provided and hack away at them. Old school! 🙌

Just to reiterate, as far as Lua is concerned, there are no differences between scripts (i.e. .lua files) and snippets. In addition, both will work equally well when run from the GUI or CLI versions of Inkspector. The only difference between the two is that snippets may not sleep or yield (because of the intention that snippets perform a task and finish rather than hang around), and if you attempt to sleep within a snippet, you will see an error message like the following:

```
inks.sleep(1)
error executing buffer : attempt to yield from outside a coroutine
```

which is Lua's way of saying it ain't sleeping.

To help mitigate against the execution of malicious scripts, Inkspector will not load script files that have been pre-compiled into Lua byte code (e.g. by using its `luac` tool). Scripts written for the Inkspector 1.0 screensaver will not work in Inkspector 2 either.

## Lua Extensions

Almost all of Inkspector's Lua extensions are contained within a table called "inks". The exceptions are:

### **print(args...)**

The standard Lua command `print` which has been replaced with Inkspector's own. This behaves the same as Lua's, except it sends its output to the Inkspector message system rather than letting it try to print to the console (which isn't there in the GUI version of course).

```
print(1,2,3,"abc",inks.status.ok)      →      1      2      3      abc      0
```

### **tprint(table)**

`tprint(table)` recursively prints the contents of the table *table*. Since many of the Inkspector script extensions return a table, this command is useful to see what's being returned or looking at one Inkspector's many pre-defined tables. For example, running the following

```
tprint(inks.edition)
```

...will display the available editions of Inkspector. The names on the left are the keys of the [inks.edition](#) table, with the value shown after the colon:

```
screensaver: 2
cli: 1
gui: 0
```

And accessing the values directly:

```
print(inks.edition.cli)      →      1
```

### **dirs (path[, {options}])**

`dirs` is a Lua iterator that iterates over the specified folder and, by default, its subfolders, returning the name of each file found. The simplest use of it takes just the name of a folder, which iterates through every file in the directory and any sub-directories it may have:

```
for path in dir([[c:\my_directory]]) do
    print(path)
end
```

Produces

```
c:\my_directory\a\file_in_a.txt
c:\my_directory\b\file_in_b.txt
c:\my_directory\c\file_in_c.txt
c:\my_directory\file.txt
```

If we only want to iterate over the files in the named directory and not in any sub-directories, we can use one of `dirs`' options: `sub_dirs`. Note all of `dirs` options are specified within a table (hence the curly braces) immediately after the name of the directory:

```
for path in dir([[c:\my_directory]],{sub_dirs=false}) do
    print(path)
end
```

Produces

```
c:\my_directory\file.txt
```

Because Inkspector scripts are particularly useful for batch snapshot conversions, or saving out .scr screenshots from snapshot files, etc, `dirs` provides the ability to filter the filenames returned. There are four options that control filtering, all specified within the table following the directory name, just as with `sub_dirs` above.

```
include_classes={file classes}
exclude_classes={file classes}
include_types={file types}
exclude_types={file types}
```

specifying `include_classes` causes `dirs` to return *only* the files that match *file classes*. The values for file classes are taken from the `inks.file_class` table, e.g.

```
inks.file_class: (10 items)
-----
archive    disk      poke      script    tape
assembly   mdrv     screenshot snapshot   unknown
```

`exclude_classes` causes files matching the specified file classes to be excluded.

Similarly, specifying `include_type` causes `dirs` to return *only* the files that match *file types*, with `exclude_types` excluding files matching the file types. The values for file types are taken from the `inks.file_type` table, e.g.

```
inks.file_type: (31 items)
-----
ace  dsk  lua  none  p81  rom  scr  snx  tap  z80  zxs
asm  e80  mdr  o     pok  rzx  slt  sp   tzx  z81
csw  e81  mlt  p     pzx  s    sna  szx  wav  zip
```

Consider a folder named [c:\ink\\_files](#) containing one each of the file types supported by Inkspector (i.e. one each with one of the 31 file extensions above)

If we only want to look for screenshot files, we would use `include_classes` as follows

```
for path in dir([[c:\ink_files]],{include_classes={inks.file_class.screenshot}})
do
    print(path)
end

→ c:\ink_files\file.mlt
  c:\ink_files\file.scr
```

You can also combine all four types of filters. Maybe for some convoluted reason, we wanted to look for all supported snapshot types, plus screenshots, we could use:

```

for path in dir([[c:\ink\_files]],{
    include_classes={inks.file_class.screenshot, inks.file_class.snapshot}})
do
    print(path)
end
→    c:\ink_files\file.ace
    c:\ink_files\file.mlt
    c:\ink_files\file.rom
    c:\ink_files\file.rzx
    c:\ink_files\file.scr
    c:\ink_files\file.slt
    c:\ink_files\file.sna
    c:\ink_files\file.snx
    c:\ink_files\file.sp
    c:\ink_files\file.szx
    c:\ink_files\file.z80
    c:\ink_files\file.z81
    c:\ink_files\file.zxs

```

However, if for some other reason we don't want .mlt, .z81 or .zxs files in the results, we would use the following to exclude them, taking care not to mix up file classes with types:

```

for path in dir([[c:\ink\_files]],{
    include_classes={inks.file_class.screenshot, inks.file_class.snapshot},
    exclude_types={inks.file_type.mlt, inks.file_type.z81, inks.file_type.zxs}
}) do
    print(path)
end
→    c:\ink_files\file.ace
    c:\ink_files\file.rom
    c:\ink_files\file.rzx
    c:\ink_files\file.scr
    c:\ink_files\file.slt
    c:\ink_files\file.sna
    c:\ink_files\file.snx
    c:\ink_files\file.sp
    c:\ink_files\file.szx
    c:\ink_files\file.z80

```

## Status Codes

Inkspector has its own set of error codes used to indicate success or otherwise. Their values are stored in the inks.status table. Probably the most useful of these values is the 'ok' value used to indicate success. Its value is stored as inks.status.ok. You can see all available status values by typing the following into the Lua bar at the top of the messages window (the output will be shown on the [messages window](#))

```
tprint(inks.status)
```

It is possible (although fairly unlikely) that the status numerical values may change between releases of Inkspector, so always compare a result with the status name rather than its actual value,

so always use, for example, `inks.status.ok` instead of 0. This also helps prevent you from falling into the trap of assuming Lua will treat the value 0 as false with such code as “if result then...”. Lua treats anything other than the boolean *false* and the special non-value *nil* as true, so the value 0 would test as true. Assuming `result` is a value returned by an inks function, the correct ways to test for success or failure are:

```
if result == inks.status.ok then
    -- success!
end
```

```
if result ~= inks.status.ok then
    -- failure!
end
```

You can see a list of Inkspector extensions at the command line by typing `incli --list`. Although this list doesn’t include a description of the functions, it does provide a useful aide-mémoire.

The remaining Inkspector Lua extensions are contained within the `inks` table.

### **inks.add\_breakpoint({parameters...})**

Creates a breakpoint, returning a unique breakpoint id if successful, otherwise 0 (which is never a valid breakpoint id). The recognised parameters are:

Parameter	Value
id	Specifies the breakpoint to amend, delete, disable or enable. Unused when creating a new breakpoint with <code>inks.add_breakpoint</code> .
type	Specifies the type of breakpoint. Must be one of the following specified as a string: mem_read, mem_write, mem_readwrite, port_read, port_write, port_readwrite, op_read, r_countdown, int_taken, di/halt, condition, int_retriggered, nmi_taken
address	Set the address of the breakpoint. It may be a number (0-65535), string (e.g. “FRAMES”) or a numeric expression. A ROM or RAM page may be specified as part of the address by preceding it with <code>RAMn:</code> or <code>ROMn:</code> ; otherwise default paging is assumed.
address_mask	Set the address mask for the breakpoint (see <a href="#">Breakpoints</a> for details)
rom	Sets the ROM associated with the address. This is the same value set with the <code>ROMn:</code> prefix for address.
ram	Sets the RAM page associated with the address. This is the same value set with the <code>RAMn:</code> prefix for address.
condition	Specifies the breakpoint’s condition.
comment	Specifies a comment for the breakpoint.
snippet	Specifies a snippet to run when the breakpoint is hit. Snippets are numbered 1 to 10, or may be specified by their description.
break_when	Specifies when a breakpoint should hit when all conditions are true. It must be one of the following specified as a string:

	always, equal_to, equal_or_greater, multiple_of, never
break_when_count	Specifies the break-when count, which is used with break_when.
one_shot	Specifies whether the breakpoint is a one-shot (deleted when hit) or not.
notify	Specifies whether the breakpoint hit is broadcast to the rest of Inkspector, which is the mechanism used to open the debugger window when one is hit. By default this value is true, which is what is desired in almost all cases. Some Inkspector test scripts that are run before a release use notify=false to prevent the debugger appearing during their run.

Examples (see the [Breakpoints](#) section for additional information on these parameters)

Break when an op-code is read from address \$0001

```
brk_num = inks.add_breakpoint{type="op_read", address=0x0001}
```

Break when the Z80 would otherwise hang with a HALT instruction performed after a DI.

```
brk_num = inks.add_breakpoint{type="di/halt"}
```

Break whenever the HL register pair has the value 49153. Note that for breakpoint types of ‘condition’, the other fields (e.g. address) are not considered.

```
brk_num = inks.add_breakpoint{type="condition", condition="hl==49153"}
```

Break when a memory write is performed between addresses 52000 and 52100 inclusive. Note that address\_mask is set to 0 so that all addresses are considered.

```
brk_num = inks.add_breakpoint{type="mem_write", address_mask=0,
condition="@addr>=52000 && @addr<=52100"}
```

Break when the value \$01 is read from address 65534

```
brk_num = inks.add_breakpoint{type="mem_read", address_mask=0,
condition="@addr==65534 && @rw==1"}
```

## inks.add\_log\_capture([filters])

Creates a new log capture that records all log entries (the messages that appear on the message window) optionally filtering them by the source of the message. An example use is a script I have that tests the speech chip ‘subtitles’ features is working as expected. It does this by capturing the log, playing some speech, then checking that the expected subtitles appear in the captured log. Here is a cut down version of the script you can try saving out to a .lua file then loading as a script file (this is one case where it won’t work as a [snippet](#), because the script needs to sleep while the BASIC command is being entered, and snippets cannot currently sleep).

The function returns a capture id, which is subsequently used to retrieve the text.

```
inks.create_machine("48k", inks.peripheral.currah_uspeech)
inks.enable_speech_subtitles(true)
- Create the log capture. AL2 is where the speech chip messages come from
capture_id=inks.add_log_capture("AL2")
```

```

inks.keyboard_assist([[LET s$="HELLO"]])
print("Waiting for BASIC command to be entered...")
while inks.is_keyboard_assist_active() do
    inks.sleep(0.5)
end
print("Done.")
-- Wait for a little longer for the BASIC command to be processed
inks.sleep(2.0)
-- Grab the captured log messages that should include the subtitles
-- no time, no header, just the text
captured = inks.get_captured_log_text(capture_id, false, false)
print("Captured:")
tprint(captured)
-- We don't need to capture anything else
inks.delete_log_capture(capture_id)

```

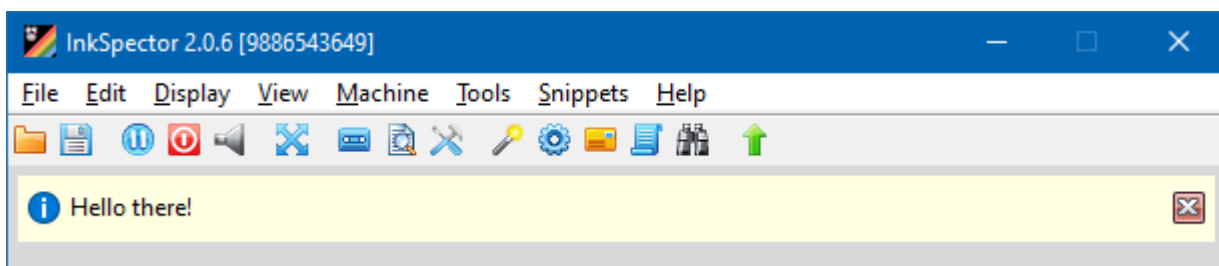
## inks.add\_message\_for\_user(message[,message type][,show-time-secs])

Queues a message for the user. When used within the GUI, the message is shown on the information bar. For the CLI, it's shown at an appropriate point (e.g. just before it closes, or during the running of a long-running process). *Message type* is an optional value from the inks.msg\_type table:

inks.msg_type.info	The default style of message (shown below in the screenshot). This is the default message type if one isn't specified.
inks.msg_type.warning	Indicates a warning message
inks.msg_type.error	Indicates an error message
inks.msg_type.success	Indicates a message bragging about something

Additional you may specify a 3<sup>rd</sup> parameter to force the message's display time in seconds, overriding the current setting.

```
inks.add_message_for_user("Hello there!")
```

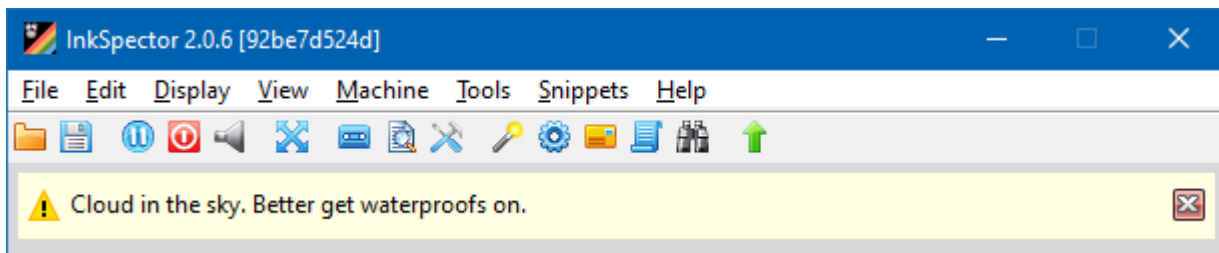


```

inks.add_message_for_user("Cloud in the sky. Better get waterproofs on.",
    inks.msg_type.warning, 30)

```





And this message would be shown for 30 seconds.

### **inks.add\_rzx\_rollback()**

Adds an RZX recording [rollback point](#) if an RZX recording is currently underway, otherwise does nothing. Returns an inks.status value.

### **inks.amend\_breakpoint(id=breakpoint\_id,[parameters...])**

Amends an existing breakpoint identified by its id.

```
brk_id = inks.add_breakpoint{type="mem_read",address=32768}  
inks.amend_breakpoint{id=brk_id,address=49153}
```

This function returns true if the breakpoint was amended. If id is missing from the parameters or is not a valid breakpoint id, false is returned. All the parameters available to [inks.add\\_breakpoint\(\)](#) may be used.

### **inks.assemble(assembly[,parameters])**

Assembles the string *assembly*, returning a status code and an elapsed-t-state value. The first value returned is the general Inkspector status code (such as those in the inks.status table). The second value returned is the number of elapsed t-states while profiling the assembled code. Unless you've set the *profile* option below to true, elapsed t-states will be zero.

Some aspects of the assembler may be controlled with the optional *parameters*, which can each be set to true or false.

trace	Enables tracing of the assembly, which can be useful when diagnosing issues with it
verbose	Enables verbose mode to show more information than usual
readonly	Enables read-only mode, where it goes through the motions of an assembly, but no files are produced. The file preview feature of Inkspector always runs the assembler in read-only mode so that trails of output files aren't left behind while previewing!
quiet	Keeps assembler output messages to a minimum
profile	When set to true, executes the assembled code, automatically stopping before executing outside of the assembled memory area, and returns the number of elapsed t-states. See below for examples.

Note that you must use the same source layout conventions as when assembling a source file (i.e. with the left-most column reserved for labels).

While you may pass assembly as a regular string, with each line separated with a newline separator, e.g. “\tORG 32768\n\tLDIR\n”, I strongly suggest using Lua’s lovely bracketed string literals if you want to keep your sanity:

```
result = inks.assemble(  
[[  
    ORG 32768  
    LD A,"I"  
    RST 10h  
    RET  
]])  
  
if result ~= inks.status.ok then  
    -- Oh dear, the assembler didn't like that. Let's see why:  
    print(inks.status_str(result))  
end
```

The assembled output is placed directly into the current machine’s memory (unless the [TARGET](#) directive is used to change the current machine type, in which case a new machine will be created). You can think of this function as a **super** POKE! You may call [inks.get\\_assembler\\_stats\(\)](#) afterwards to return a table containing some statistics on the last use of the assembler.

If the *profile* parameter is specified and set to true, the assembled code is executed, automatically stopping when execution goes outside of the assembled area, and the number of t-states the execution took is returned as the 2<sup>nd</sup> value, the result of the assembly being the first. For example:

```
result, elapsed = inks.assemble(  
[[  
    org    32768  
    ld     b,c  
    ldi  
    ld     de,6000h  
]],{profile=true})  
print("This code took "..elapsed.." ts to execute")
```

→ This code took 30 ts to execute

Note that the profiling automatically stops when the execution goes outside of the assembled area of memory. So, for example, adding a `rst 0` to the above example...

```
result, elapsed = inks.assemble(  
[[  
    org    32768  
    ld     b,c  
    ldi  
    ld     de,6000h  
    rst    0  
    ; Profiling never reaches here because the rst executes code at address 0  
    ld     bc,0  
    ld     hl,0  
    ld     de,0  
    ldir  
]],{profile=true})  
print("This code took "..elapsed.." ts to execute")
```

→ This code took 41 ts to execute

To avoid the profiler getting stuck waiting for the code to finish, it is abandoned if it's not reached the end of the assembled area of memory after one seconds' worth of real-time emulation. The following demonstrates this:

```
result, elapsed = inks.assemble(
[[
    org    32768
loop: jr    loop
    ; Profiling never reaches here because of the endless loop above
    ld     bc,0
    ld     hl,0
    ld     de,0
    ldir
]],{profile=true})
print("This code took "..elapsed.." ts to execute")
```

→ This code took 0 ts to execute

Note how the elapsed number of t-states is 0 to indicate it being abandoned.

## **inks.assemble\_file(file[,parameters])**

As [inks.assemble\(assembly\)](#), but takes an assembler filename as its argument. The optional parameters work exactly the same as they do for `inks.assemble`.

```
result = inks.assemble_file([[c:\temp\little_function.s]],{profile=true})
```

You may be wondering why this function exists as it's pretty similar to using [inks.load\(some\\_file.s\)](#). Some Inkspector test scripts use this function to perform profiling tests.

## **inks.assembler\_eval(expression)**

Passes the *expression* to the assembler's own expression evaluator (which is separate to the main one used by Inkspector's debugger) and returns the result, or nil if the expression couldn't be evaluated. If this function is used before the assembler has, it will return nil as the assembler is not created until it's used.

```
print(inks.assembler_eval("FRAMES"))    → nil (assuming the assembler hasn't been
used first)
```

```
inks.assemble([[
    TARGET 48K
    IMPORT SYSVARS
    ORG 32768
Start:
]])
print(inks.assembler_eval("FRAMES"))    →      23672
print(inks.assembler_eval("Start"))      →      32768
```

## **inks.attach\_peripheral(peripheral[,peripheral])**

Performs a hot attachment (i.e. the machine is not rebooted afterwards) of one or more peripherals. The peripherals are specified by using the values from the `inks.peripheral` table (you can run the

CLI with `inclcli --listconstants` to see the available values). The function returns the number of peripherals attached.

```
print(inks.attach_peripheral(inks.peripheral.kempston_joystick)) → 1
print(inks.attach_peripheral(inks.peripheral.kempston_joystick)) → 0 (already
attached)
```

## **inks.check\_mdrv\_cart([unit-num])**

Performs some file integrity tests on the cartridge inserted into Microdrive #1 (or *unit-num* if specified), returning a status value and the number of errors found.

```
inks.insert_mdrv_cart([c:\temp\demo.mdr])
result, num_errors = inks.check_mdrv_cart()
print(inks.status_str(result).."": "..num_errors.." errors found")
```

## **inks.clear\_captured\_log\_text(capture-id)**

Clears any text captured by the capturer identified by *capture-id*.

```
capture_id = inks.add_log_capture()
-- ...Do some things...
-- Then clear any text captured so far
inks.clear_captured_log_text(capture_id)
```

## **inks.clear\_ram([initialisation-value])**

Sets the memory contents of the current machine to *initialisation-value*, or zero if not specified.

```
inks.clear_ram()    -- Clear all memory to value 0
inks.clear_ram(255) -- Clear all memory to value 255
```

## **inks.compare\_lines(lines1, lines2)**

Compares two strings, assumed to be containing multiple lines each. This function returns six values: result, numLines1, numLines2, firstLineMismatch, s1MismatchLine, s2MismatchLine.

*Result* is one of the following values:

0	lines1 is equal to lines2
1	lines1 and lines2 contain a different number of lines
2	Lines1 and lines 2 contain the same number of lines, but there's at least one line mismatch

This function is not intended to be particularly useful for others, but it's used in the spooling test scripts in a similar way to this:

```
local cresult, nl1, nl2, nlmismatch, s1line, s2line =
    inks.compare_lines(expected_text, spooled_text)
if cresult == 0 then
    -- 0 = match
    print("match:", nl1.." lines", nl2.." lines")
elseif cresult == 1 then
    -- 1 = different number of lines
    print("unequal # lines: ", nl1.." lines", nl2.." lines")
else
    -- 2 = mismatch at line nlmismatch
```

```

        print("line difference: ", "line "..nlmismatch..": "..s1line, "line
"..nlmismatch..": "..s2line)
    end

```

## inks.compress\_basic\_lines(lines)

Given a multiline string *lines* containing ASCII BASIC commands, this function returns a string with multiple spaces and backslashes stripped. Like [inks.compare\\_lines](#), this is a niche function used specifically by the spooling test scripts and not expected to be particularly useful elsewhere.

## inks.crc32(string)

Returns the CRC32 value from the contents of *string*.

```
print(inks.crc32("HELLO")) → c1446436
```

## inks.crc32\_from\_memory(address, length)

Returns the CRC32 value from the memory contents of the current machine starting at address *address* for a *length* number of bytes.

```
print(inks.crc32_from_memory(0,16384)) → ddee531f (for the standard 48K Spectrum ROM)
```

## inks.create\_disk(drive-num, capacity)

Creates a new disk image and inserts it into *drive-num*. Capacity is specified in KB and may be either 180 (compact floppy disc single head drive) or 720 (compact floppy disc double head drive).

## inks.create\_machine(machine-name[,peripherals])

Selects the specified machine type from a long or short description, optionally adding one or more peripherals or specifying none at all (overriding the current peripheral configuration for the machine type).

Long form	Short Form
ZX Spectrum 16k	16k
ZX Spectrum 48k	48k
ZX Spectrum 128	128k
ZX Spectrum +2	plus2
ZX Spectrum +2A	plus2a
ZX Spectrum +3	plus3
ZX80	ZX80
ZX81	ZX81
Jupiter Ace	ace

Create a 48K machine adding any peripherals as configured on the Peripherals options page for the 48K Spectrum:

```
inks.create_machine("48k")
```

Create a 48K machine with no added peripherals, even if there are some configured on the Peripherals options page:

```
inks.create_machine("48k", inks.peripherals.none)
```

Create a 48K machine with only Kempston Joystick attached, regardless of how peripherals are configured on the Peripherals options page:

```
inks.create_machine("48k", inks.peripheral.kempston_joystick)
```

Create a 48K machine with only Kempston Joystick and Fuller Orator attached, regardless of how peripherals are configured on the Peripherals options page:

```
inks.create_machine("48k", inks.peripheral.kempston_joystick,  
inks.peripheral.fuller_orator)
```

### **inks.create\_mdrv\_cart({*config*},[*Microdrive-unit-num*])**

Creates a new Microdrive cartridge image and inserts it into the specified, or default, Microdrive unit using the specified *config* details within a table. If no *Microdrive-unit-num* is specified, 1 is assumed. An inks.status code is returned indicating whether the operation succeeded.

The recognised *config* table items are:

<b><i>Config</i> item (default values in brackets)</b>	<b>Effect</b>
path (no default value – i.e. don't save the generated cartridge image to a file – use <a href="#">inks.save_mdrv_cart_as()</a> to save it to a file)	Specifies the filename that the generated Microdrive cartridge image should be saved as
cartname (yyyy-mm-dd i.e. the current date)	Specifies the name of the cartridge when it's formatted. This must be no longer than 10 characters.
sectors (180 i.e. 90kb)	Specifies the size of the generated Microdrive cartridge image in 512 byte sectors, with the minimum being 128 (64kb) and maximum 254 (127kb)
formatter (quick)	Specifies how the new cartridge image should be formatted: <b>none</b> – don't format. It will remain unusable until it's formatted. <b>quick</b> – Inkspector will format it and it will be usable by the Interface 1 immediately. <b>command</b> – Auto-type the FORMAT command once the image has been created to allow the Interface 1 to format it <b>command_without_enter</b> – as 'command' but ENTER is not pressed at the end of the command, to allow the line to be edited first
cat (false)	If true, the CAT command is run after formatting to display file list (which will be empty) and available capacity.

<code>add_rw_marker (false)</code>	If true, the created image has an extra byte tagged on the end to indicate whether the image is read-only or read-write. Inkspector sets the marker to read-write.
------------------------------------	--

```
result = inks.create_mdrv_cart({sectors=180, formatter="command", cat=true,
cartname="cat-test"}, 1)
```

## **inks.date\_as\_10\_digit\_string()**

Returns the current date as a 10-digit string in the format yyyy-mm-dd.

```
print(inks.date_as_10_digit_string()) → 2025-02-02
```

## **inks.dbg\_expand(string)**

This is used to test the Inkspector Lua preprocessor used to expand the debugger's variables directly within Lua script.

```
inks.dbg_expand("print (@pc)") → print(0)
```

## **inks.dbg\_get\_show\_hex()**

Returns true when Inkspector is configured to show hexadecimal values everywhere, otherwise false.

## **inks.dbg\_set\_last\_access(address, rw-value)**

Sets the last address and read/write values recorded by the debugger (the values exposed by the [@addr](#) and [@rw](#) pseudo variables). This is used by debugger test scripts and not expected to be particularly useful elsewhere.

## **inks.dbg\_show\_hex(show-hex-values)**

Sets Inkspector's global 'show hexadecimal values' state. For the GUI, all open windows, such as the debugger, will automatically redraw their contents if they're affected by the change. The value of the previous global hex state is returned.

```
-- Toggle the global show hex / decimal numbers setting
inks.dbg_show_hex(not inks.dbg_get_show_hex())
```

## **inks.default\_profile\_name()**

Returns the name of the default controller profile.

```
print(inks.default_profile_name()) → Default
```

## **inks.delete\_all\_breakpoints()**

Deletes all breakpoints.

That was an easy one to document.

## **inks.delete\_breakpoint(id)**

Deletes the breakpoint identified with an id of *id*, returning true if the breakpoint was found, otherwise false.

```
brk_id = inks.add_breakpoint{address=32768}
inks.delete_breakpoint(brk_id)    -- What a waste of time
```

## **inks.delete\_rzx\_recording()**

Deletes any RZX recording if one is in progress, otherwise does nothing.

## **inks.delete\_user\_messages()**

Deletes any messages that are queued up waiting to be presented to the user.

## **inks.detach\_peripheral(peripheral[,peripheral[])**

Performs a hot detachment (i.e. the machine is not rebooted afterwards) of one or more peripherals. The peripherals are specified by using the values from the inks.peripheral table (you can run the CLI with `inclu --listconstants` to see the available values). The function returns the number of peripherals detached.

```
inks.attach_peripheral(inks.peripheral.kempston_joystick)
print(inks.detach_peripheral(inks.peripheral.kempston_joystick)) → 1
print(inks.detach_peripheral(inks.peripheral.kempston_joystick)) → 0 (already
detached)
```

## **inks.did\_recording\_play\_successfully()**

Returns true if the current RZX recording played to the end successfully (i.e. didn't lose sync, etc.), otherwise false. Loading another snapshot will reset the successful playback state and return false until another RZX is played successfully to completion.

## **inks.disassemble\_to\_file(out-file, start, end, [addresses[, generate-labels]])**

Disassembles the memory contents of the current machine to a file named **out-file**, starting at address **start**, ending at **end**. If **addresses** is specified and is true, an address column is included in the disassembly. If **generate-labels** is specified and is true, labels are generated and used in the disassembly. `inks.status.ok` is returned if the disassembly generation was successful.

```
inks.load("My_fave_game_that_is_deffo_legal_to_possess.szx")
result = inks.disassemble_to_file([c:\temp\fave_game.s], 32768, 16384, true,
true)
```

## **inks.dofile(in-file)**

Executes the Lua script file **in-file**. This differs from the Lua's own `dofile` function by making the **in-file** path relative to the path of the script that is currently running (if any).

Consider a script file `c:\testers\all.lua` whose contents are:

```
inks.dofile("assembler.lua")
```

`inks.dofile` will ensure that the path is interpreted as `"c:\testers\assembler.lua"`, not just `"assembler.lua"`, which is useful when running a Lua script that needs to call other Lua scripts.



## **inks.dreamysleepynightiesnoozysnooze(time-to-sleep-for)**

Causes the current script to sleep for **time-to-sleep-for** seconds.

```
-- sleep for one and a half seconds. Dunno why.
inks.dreamysleepynightiesnoozysnooze(1.5)
```

Alternatively, you may use [inks.sleep\(\)](#) which is a much more sensibly named alias for this.

## **inks.dump\_profiles()**

Dumps the current keyboard and controller profiles to the message window (GUI) or console (CLI).

## **inks.eject\_disk(drive-num)**

Ejects any disk image that may be present in drive *drive-num*.

## **inks.eject\_mdrv\_cart([unit-num])**

Ejects any cartridge inserted into Microdrive #1 (or *unit-num* if specified).

## **inks.eject\_tape()**

Ejects any tape currently inserted.

## **inks.enable\_breakpoint(id[,enable])**

Enables the breakpoint with an id of *id*, returning true if the breakpoint was found, otherwise false. If the value *enable* is supplied (true or false) it controls whether the breakpoint is to be enabled (true) or disabled (false).

```
brk_id = inks.add_breakpoint{type="mem_read",address=32768}
inks.enable_breakpoint(brk_id, false)  -- Disable the breakpoint
inks.enable_breakpoint(brk_id)        -- Enable it again. This is fun!
```

## **inks.enable\_canvas(enable-canvas)**

Enables (true) or disables (false) writes to the canvas. At the end of each emulated frame, the native screen memory is converted to a common canvas format before being made available to the GUI and .gif creator. This function exists solely so I can monkey around turning the native memory → canvas conversion off which allows me to profile the code a little easier. So basically, not a very useful function for anyone else.

## **inks.enable\_speech\_subtitles([true|false])**

Enables or disables the showing of the phonemes being played by any of the emulated speech chips such as the one in the Currah Microspeech. If no parameter is specified, true is assumed. The previous value of this setting is returned.

```
prev_show_subtitles = inks.enable_speech_subtitles()
--
-- Do something...
```

```
--
-- Restore the setting
inks.enable_speech_subtitles(prev_show_subtitles)
```

## inks.enable\_sysvar\_symbols(enable)

Enables (true) or disables (false) the exposure of the system's variable names to the expression evaluator. Returns whether the symbols were enabled before the call to this function. Again, probably not of much use to most, but it is used in Inkspector test scripts.

```
inks.enable_sysvar_symbols(true)
print(inks.eval("FRAMES"))      → 23672
inks.enable_sysvar_symbols(false)
print(inks.eval("FRAMES"))      → nil
```

## inks.eval(expr)

Returns the result of the expression **expr** as an integer, or nil if the expression couldn't be evaluated. This function provides access to the debugger's own expression evaluator which resolves [pseudo variables](#) (which always start with a "@"), [Z80 registers](#), any symbols that may be active in the debugger and the names of system variables for the currently running system. NB this function may be used at any time; the debugger window does not need to be open, or have been opened previously.

```
print(inks.eval("1230+4")) → 1234
print(inks.eval("hl+de")) → 4229
```

## inks.eval\_watch(expr[,watch-format-specifier])

Evaluates the expression **expr**, returning the result as a string and not a number as [inks.expr](#) does, optionally formatted using the Watch window's format specifiers. The numeric result is also returned as the second parameter, with nil indicating an error, in which case the first parameter may explain the reason for failure.

```
print(inks.eval_watch("1230+4,x")) → $04D2      1234
print(inks.eval_watch("hl+de,b")) → 0b1011100101101111 47471
print(inks.eval_watch("rhubarb")) → Unknown symbol  nil
```

## inks.expand\_basic\_command(command)

Expands a BASIC command. This function is used in the Inkspector test scripts for the Key Assist feature. The first value returned is a status code with the second being the expanded string.

```
print(inks.expand_basic_command("PR.123")) → 0, PRINT 123 (0 being
inks.status.ok)
```

## inks.export\_zx\_printer(path)

Exports the ZX Printer's paper roll to a .pbm (portable bitmap) format image file, returning an inks.status value. The .pbm file created is a P1 plain text type, which makes it useful for writing testing scripts, being able to compare the exported printed output using text comparison. In fact, that's exactly why I added it!

```
status = inks.export_zx_printer([[c:\temp\out.pbm]])
```

## **inks.file\_extension(path)**

Returns the extension part of the supplied path including the dot.

```
print(inks.file_extension([[c:\temp\mayhem.szx]])) → .szx
```

## **inks.file\_name(path)**

Returns the filename part of the supplied path

```
print(inks.file_name([[c:\temp\mayhem.szx]])) → mayhem.szx
```

## **inks.file\_crc32(path)**

Returns the result code of the operation and the CRC32 value of the file *path* as an 8 character lower case hexadecimal string. The result code may be used to determine why the operation failed if its value is other than inks.status.ok.

```
print(inks.file_crc32([[c:\temp\mayhem.szx]])) → 0 5240ff8d
```

With the 0 being the result code for inks.status.ok, followed by the CRC32 value produced from the contents of the file.

## **inks.file\_sha1(path)**

Returns the result code of the operation and the SHA-1 digest of the file *path* as a 40 character lower case hexadecimal string. The result code may be used to determine why the operation failed if its value is other than inks.status.ok.

```
print(inks.file_sha1([[c:\temp\mayhem.szx]])) → 0  
695d7413d35f1dd22365e293b466bee0b872bb4e
```

With the 0 being the result code for inks.status.ok, followed by the SHA-1 digest.

## **inks.file\_size(path)**

Returns the size of the specified file, or -1 if there was an error.

```
print(inks.file_size([[c:\temp\mayhem.szx]])) → 20641
```

## **inks.file\_stem(path)**

Returns the filename stem of the supplied path

```
print(inks.file_stem([[c:\temp\mayhem.szx]])) → mayhem
```

## **inks.files\_in\_library()**

Returns three numbers regarding the number of supported files in the library folders, as configured in the GUI (Options → Library)

```
print(inks.files_in_library()) → 41 41 0
```

The values are, in the order they're returned:

1. The total number of supported files in the library folders
2. The number of files added to the library since the last time it was changed

3. The number of files removed since the last time the library folders were changed

## **inks.find\_in\_library(path)**

Searches for the given filename in the [library](#) using the same syntax as the [Inkspector search window](#), returning all matches in a table:

```
results=inks.find_in_library([[mayhem ext:rxz]])
tprint(results)

→ 1    C:\Projects\Inkspector2\Snapshots\rzx\mayhem.rzx
→ 2    D:\Emulator\Sinclair\RZX\Allowed\mayhem.rzx
```

## **inks.flush\_user\_messages()**

Flushes (i.e. displays immediately) all user messages currently queued. This only works for the CLI. It does nothing when executed within the GUI.

## **inks.get\_active\_mdrv()**

Returns the number of the currently active Microdrive unit (1-8). If none are active, 0 is returned. If there is no Interface 1 attached, -1 is returned.

```
inks.create_machine("48k", inks.peripheral.none)
print(inks.get_active_mdrv())                →    -1
inks.create_machine("48k", inks.peripheral.zx_if1)
print(inks.get_active_mdrv())                →     0
```

## **inks.get\_all\_breakpoints()**

Returns a list of all breakpoints set for the current machine in a table. If you add two breakpoints in the debugger, then run the following snippet

```
tprint(inks.get_all_breakpoints())
```

You will see something along the following lines in the message window. Note the index into the table (1 and 2) are just to hold each breakpoint entry and are not the id of the breakpoints. The breakpoint ids are shown as the *id* value:

```
1:                                     ← Not the breakpoint id
    snippet: 0
    type: op_read
    condition:
    address: SPEC:$15EF
    break_when_count: 1
    system: false
    comment:
    id: 1                               ← This is the breakpoint id
    hit_count: 0
    enabled: true
    break_when: always
    pc: 65535
    one_shot: false
    symbol_refs:
    notify: true
    pause: true
    address_mask: $FFFF
```

2:

```
snippet: 0
type: mem_read
condition:
address:  SPEC:$1610
break_when_count: 1
system: false
comment:
id: 2
hit_count: 0
enabled: true
break_when: always
pc: 65535
one_shot: false
symbol_refs:
notify: true
pause: true
address_mask: $FFFF
```

## **inks.get\_all\_model\_names([long\_names])**

Returns a table containing the names of all supported machines, by default using the short versions. Specify true to retrieve the long versions.

```
for k,v in ipairs(inks.get_all_model_names(false)) do
    print(k,v)
end

1      16k
2      48k
3      128k
4      +2
5      +2A
6      +3
7      ZX80
8      ZX81
9      ACE
```

```
for k,v in ipairs(inks.get_all_model_names(true)) do
    print(k,v)
end

1      ZX Spectrum 16k
2      ZX Spectrum 48k
3      ZX Spectrum 128
4      ZX Spectrum +2
5      ZX Spectrum +2A
6      ZX Spectrum +3
7      ZX80
8      ZX81
9      Jupiter ACE
```

## **inks.get\_all\_model\_nums()**

Returns a table containing the inks.model values of all supported machines.

```
for k,v in ipairs(inks.get_all_model_nums()) do
    print(k,v,inks.get_model_name(v, true))
end
```

1	0	ZX Spectrum 16k
2	1	ZX Spectrum 48k
3	2	ZX Spectrum 128
4	3	ZX Spectrum +2
5	4	ZX Spectrum +2A
6	5	ZX Spectrum +3
7	6	ZX80
8	7	ZX81
9	8	Jupiter ACE

## **inks.get\_attempted\_load\_file\_class()**

Returns the file class of the most recently attempted file load (i.e. whether it succeeded or not).

```
result = inks.load([[horizons.tzx]])
if result == inks.status.ok then
    fc = inks.get_attempted_load_file_class()
    print("File class: "..inks.get_file_class_str(fc))
end

→      File class: tape
```

## **inks.get\_assembler\_stats()**

Returns a table containing the statistics for the most recent assembler activity.

```
inks.load([[Inkspector2\Snapshots\s\example.s]])
stats = inks.get_assembler_stats()
for k,v in pairs(stats) do
    print (k..": "..tostring(v))
end
```

Would print something along the lines of:

```
exec_mode: 1
program_start_address: 32886
exec_start_time: 0
exec_mode: 1
num_lines: 432
assemble_time: 0.0062206001020968
num_instructions: 122
address: 49152
num_undoc_instructions: 6
```

## **inks.get\_attached\_peripherals()**

Returns a table indicating which peripherals are currently attached to the machine-name

```
periphs = inks.get_attached_peripherals()
for k,v in ipairs(periphs) do
    print(k,inks.peripheral_name(v))
end
```

Would print something along the lines of:

1	Kempston Joystick Interface
2	Fuller Orator

## inks.get\_base\_folder()

Return the name of Inkspector's base folder – i.e. the folder its executable is running from.

```
print(inks.get_base_folder()) → C:\Program Files\Inkland\InkSpector 2
```

## inks.get\_basic\_program([line-nums-as-keys[,show-hidden-numbers]])

Attempts to list the BASIC program held in the current machine, returning the results as a table containing one or more sub-tables that contain the BASIC program and, separately, any user variables currently defined in the system. By default the keys used in the table are a counter from 1 to number of lines of BASIC program so that any weird and wonderful programs that have poked fruity line numbers as anti-hacker measures do not mess up the ordering of the lines as they're stored. If you do want the keys to be the line numbers, pass in *line-nums-as-keys* as true, but be aware of the possible complications mentioned.

When this command is executed with the Jupiter ACE present, it returns all Forth words currently defined, grouped into whether they're in ROM or RAM.

If you spool the following into a 48K Spectrum:

```
10 REM Example
20 LET A=1
```

Then then run the following snippet:

```
result, basic = inks.get_basic_program()
tprint(basic.blocks[1])
```

Will display

```
→ vars:
→ program:
→      1:  10 REM Example
→      2:  20 LET A=1
```

Note how under “program” the key in the table is a value that is incremented from 1 (rather than being the actual line number), so that the values – the lines of BASIC – appear in the order they're defined. If you pass ‘true’ as the first value to `get_basic_program` the output will use the line numbers as keys, producing:

```
→ program:
→      10:  10 REM Example
→      20:  20 LET A=1
→ vars:
```

Which is fine here in this simple example, but if there's some anti-hacker tomfoolery going on with the line numbers, the lines might appear in an order different to the one they're defined.

### Lua Note

Because the keys in the second snippet are now line numbers (10, 20, etc.), and not sequential integers (1, 2, etc. as required by the Lua iterator `ipairs`) as in the first one, we have to use pairs (not `ipairs`). Incidentally, pairs could be used in the first snippet instead of `ipairs`, but then the ordering of the values is no longer guaranteed.

## **inks.get\_basic\_program\_from\_tape(line-nums-as-keys)**

As [inks.get\\_basic\\_program](#), but looks for BASIC programs in all blocks of the currently loaded tape image. This feature currently supports .tap, .tZX (standard speed data block) and .pZX tape images.

```
result, basic = inks.get_basic_program_from_tape()
for k,v in ipairs(basic) do
    print (v)
end
```

## **inks.get\_basic\_programs\_from\_mdrv(line-nums-as-keys[,unit-num])**

As [inks.get\\_basic\\_program](#), but looks for BASIC programs in all files on Microdrive unit 1 (unless overridden with *unit-num*).

```
result, basic = inks.get_basic_programs_from_mdrv()
for k,v in ipairs(basic) do
    print (v)
end
```

## **inks.get\_breakpoint(id)**

If *id* is a valid breakpoint id, returns a table describing the breakpoint otherwise returns nil.

```
brk_id = inks.add_breakpoint{type="condition", condition="hl==49153"}
bp = inks.get_breakpoint(brk_id)
for k,v in pairs (bp) do
    print(k,v)
end
```

Produces (NB the order may be different)

symbol_refs	
address	UNMAP:\$0000
system	false
enabled	true
pause	true
hit_count	0
comment	
break_when	always
one_shot	false
notify	true
type	condition
break_when_count	1
address_mask	\$FFFF
condition	hl==49153
snippet	0
id	1
pc	65535 ← The value of the Z80's PC register when hit

## **inks.get\_breakpoint\_ids()**

Returns a table containing the ids of currently active breakpoints.



```
ids = inks.get_breakpoint_ids()
for k,v in ipairs(ids) do
    print("Breakpoint #"..k.." has an id of "..v)
end
```

Would print something along the lines of:

```
Breakpoint #1 has an id of 1
Breakpoint #2 has an id of 2
```

## **inks.get\_captured\_log\_text(capture-id)**

Returns the text captured by the capturer with the specified id.

See [inks.add\\_log\\_capture\(filters\)](#)

## **inks.get\_cwd()**

Returns the current working directory

```
print(inks.get_cwd())    →    C:\Program Files\Inkland\InkSpector 2
```

## **inks.get\_disk\_image\_path(drive-num)**

Returns a status and the path to the disk image file loaded into *drive-num*.

If *drive-num* doesn't exist, inks.status.hardware\_no\_present and nil are returned.

If *drive-num* exists but there's no disk 'inserted' into it, inks.status.no\_media and nil are returned.

If *drive-num* exists and there's a disk loaded in it, inks.status.ok and the path of the image are returned. The path will be empty if the disk image has been created but not yet saved to a file.

## **inks.get\_edition()**

Returns the edition of Inkspector being run. The returned value is one of the following values in the inks.edition table:

inks.edition.gui
inks.edition.cli
inks.edition.screensaver

```
print("The editions are:")
tprint(inks.edition)
print("Currently running edition "..inks.get_edition())
```

Outputs:

```
The editions are:
screensaver: 2
cli: 1
gui: 0
Currently running edition 0
```

## inks.get\_execution\_state()

Returns the Z80's execution state. The value will be one of the following:

Value	Meaning
inks.z80_state.running	The Z80 is running.
inks.z80_state.exhausted	The Z80 has finished running all the t-states it's been asked to executed.
inks.z80_state.ddfd_loop	The Z80 encountered 64K's worth of contiguous DD/FD opcodes
inks.z80_state.exit_requested	A request has been made for the Z80 to stop executing. This is usually due to a breakpoint or an emulation event about to occur.
inks.z80_state.r_countdown_zero	The Z80 has finished running for a number of R-register changes. This is used by .rzx playback on the ZX Spectrums.

## inks.get\_execution\_state\_str([execution\_state])

Returns a string description for the current execution state, or a specific one if a value is passed in.

```
print(inks.get_execution_state_str())           →    exhausted
print(inks.get_execution_state_str(inks.z80_state.ddfd_loop)) →    ddfd_loop
```

## inks.get\_file\_class(path)

Returns the file class determined from the *path*. The value returned is one in the inks.file\_class table shown below.

Value	File Would Be...	File Extensions
inks.file_class.assembly	Assembled	.asm .s
inks.file_class.disk	Loaded into the Spectrum +3's disk drive	.dsk
inks.file_class.mdrv	Loaded into an Interface 1 Microdrive	.mdr
inks.file_class.poke	Loaded into the POK manager	.pok
inks.file_class.screenshot	Loaded as a screenshot, and the machine paused so it may be viewed	.scr .mlt
inks.file_class.script	Loaded and executed as a Lua script with Inkspector extensions available.	.lua
inks.file_class.snapshot	Loaded as a snapshot into a suitable machine as specified by the snapshot type	.sna .sna .z80 .slt .rzx .szx .sp .zxs .ace

inks.file_class.tape	Loaded into the cassette player	.tap .csw .tzx .pzx .wav .80 .o .81 .p .p81
inks.file_class.archive	Assuming a supported file is found within it, it would be treat as the supported file class.	.zip
inks.file_class.unknown	Not a direct file class supported by Inkspector	Anything not shown above

```
print(inks.get_file_class("test.szx")) → 1
```

(1 being the value for inks.file\_class\_snapshot)

## inks.get\_file\_class\_str(file\_class)

Returns a string description for the specified file\_class, or nil if file\_class is invalid.

```
fc = inks.get_file_class("test.szx")
print(fc, inks.get_file_class_str(fc)) → 1 snapshot
```

## inks.get\_file\_type(path)

Returns the file type determined from the *path*. The value returned is one from the inks.file\_type table:

Value	Extension
inks.file_type.sna	.sna
inks.file_type.szx	.SZX
inks.file_type.z80	.z80
inks.file_type.slt	.slt
inks.file_type.rzx	.rzx
inks.file_type.scr	.scr
inks.file_type.mlt	.mlt
inks.file_type.szx	.szx
inks.file_type.rom	.rom
inks.file_type.sp	.sp
inks.file_type.tap	.tap
inks.file_type.csw	.csw
inks.file_type.tzx	.tzx
inks.file_type.pzx	.pzx
inks.file_type.wav	.wav
inks.file_type.dsk	.dsk
inks.file_type.e80	.80

inks.file_type.o	.o
inks.file_type.e81	.81
inks.file_type.p	.p
inks.file_type.p81	.p81
inks.file_type.ace	.ace
inks.file_type.mdr	.mdr
inks.file_type.pok	.pok
inks.file_type.lua	.lua
inks.file_type.s	.s
inks.file_type.asm	.asm
inks.file_type.zxs	.szx

## inks.get\_frame\_tstates()

Returns the number of elapsed t-states for the current emulation frame.

## inks.get\_full\_address(addr-str | addr[,ram,rom])

Expands an address, specified as either a string or separate address, ram, rom values into separate address, ram, rom values. The values returned are corrected as per the state of the machine at the time. That is, if specifying a ROM page when the address is in RAM (or vice versa), the returned ROM/RAM values will reflect what the address represents.

The values are returned in address, RAM, ROM order, with nil indicating “not a ROM or RAM”. The address is a 16-bit number, RAM is a value from the inks.ram table, and ROM a value from the inks.rom table.

When run on a 16k Spectrum, the following is returned:

```
print(inks.get_full_address(0))      →      0      nil      0
print(inks.get_full_address(16384))  →     16384  5      nil
print(inks.get_full_address(49152))  →     49152 nil      nil
```

The 0 at the end of the first line is the value inks.rom.zx\_48k, indicating the standard 16k/48k ROM is present at the specified address of 0. The RAM page is returned as nil for address 49152 as that address is unmapped for a 16K Spectrum, with its memory ending at 32767.

When run on a 48k Spectrum, the RAM page is returned as inks.ram.page\_0 instead of nil as there is memory there:

```
print(inks.get_full_address(49152))  →     49152  0      nil
```

If we try and request a RAM page that doesn’t exist on that machine, the actual RAM page occupying the specified address is returned:

```
print(inks.get_full_address(49152, inks.ram.page_6)) →     49152  0      nil
```

Things get (slightly) more interesting when running on a 128k Spectrum:

```
print(inks.get_full_address(49152))           →    49152  7    nil
print(inks.get_full_address(49152, inks.ram.page_6)) →    49152  6    nil
```

Note how we've been able to specify RAM page 6 which does exist on the 128k model, overriding the current RAM page 7 active at address 49152.

## inks.get\_full\_address\_from\_str(addr-str)

Extracts address, RAM and ROM values from a string-format address. Unlike [inks.get\\_full\\_address\(\)](#) this function only extracts the values from the string and does not attempt to validate them against the currently running machine. The values returned are a boolean indicating whether operation succeeded or not, followed by address, RAM (from the inks.ram table) and ROM (from the inks.rom table) values.

```
print(inks.get_full_address_from_str("32768"))      → true 32768 nil    nil
print(inks.get_full_address_from_str("RAM04:49152")) → true 49152 4     nil
print(inks.get_full_address_from_str("ZXIF1:1024")) → true 1024 nil    5
print(inks.get_full_address_from_str("rhubarb"))     → false 0    nil    nil
```

The third line is specifying address 1024 within the ZX Interface 1 ROM (inks.rom.zx\_if1 with the value 5).

The full list of recognised RAM page and ROM names are as follows:

RAM Page Name	Represents
RAM00	RAM page 0
RAM01	RAM page 1
RAM02	RAM page 2
RAM03	RAM page 3
RAM04	RAM page 4
RAM05	RAM page 5
RAM06	RAM page 6
RAM07	RAM page 7
RAMMF	Multiface RAM
RAMCH	Chroma 81 RAM

ROM Name	Represents
SPEC	ZX Spectrum 16k/48k ROM
128R0	ZX Spectrum 128 ROM 0
128R1	ZX Spectrum 128 ROM 1
+2R0	ZX Spectrum +2 ROM 0
+2R1	ZX Spectrum +2 ROM 1
ZXIF1	ZX Interface 1 ROM

ZXIF2	ZX Interface 2 ROM
ZX80	ZX80 ROM
ZX81	ZX81 ROM
+3R0	ZX Spectrum +3 ROM 0
+3R1	ZX Spectrum +3 ROM 1
+3R2	ZX Spectrum +3 ROM 2
+3R3	ZX Spectrum +3 ROM 3
ACE	Jupiter ACE ROM
USPCH	Currah uSpeech ROM
USRC	Currah uSource ROM
MF1	Multiface 1 ROM
MF128	Multiface 128 ROM
MF3	Multiface 3 ROM
SPCMT	Spec-Mate ROM

## **inks.get\_full\_address\_str(addr-str | addr[,ram,rom])**

Returns a string-format address (such as one accepted by [inks.get\\_full\\_address\\_from\\_str\(\)](#)) from either a string or numeric format address. This function does use the current state of the machine to validate and override RAM and ROM page values. Note this function is also affected by the current state of the global hex value state.

When run on a 16k Spectrum:

```
print(inks.get_full_address_str(32768))    →    UNMAP:$8000
```

When run on a 48k Spectrum:

```
print(inks.get_full_address_str(32768))    →    RAM02:$8000
```

## **inks.get\_last\_breakpoint\_hit()**

Returns a table containing the details of the last breakpoint hit in the same format as [inks.get\\_breakpoint\(\)](#). If a breakpoint hasn't yet been hit, the reported id (`lastbrk.id` in the example below) will be 0. 0 is never a valid breakpoint id.

```
lastbrk = inks.get_last_breakpoint_hit()
print("PC at last hit: "..lastbrk.pc)

→    PC at last hit: 32768
```

## **inks.get\_last\_spool\_result()**

Returns the `inks.status` value set by the most recent spooling operation.

## **inks.get\_machine\_model()**

Returns the current machine model as a value from the `inks.model` table.

## **inks.get\_machine\_state()**

Returns the complete machine state as a table of strings.

```
mstate = inks.get_machine_state()
for k,v in ipairs(mstate) do
    print(v)
end
```

## **inks.get\_mdrv\_cart\_image\_path([unit-num])**

Returns the file path of the cartridge image inserted into Microdrive #1 (or *unit-num* if specified). If the Microdrive unit doesn't exist or doesn't have a cartridge in it, an empty path is returned.

```
print(inks.get_mdrv_cart_image_path())      →
inks.insert_mdrv_cart([[c:\temp\demo.mdr]])
print(inks.get_mdrv_cart_image_path())      →      c:\temp\demo.mdr
```

## **inks.get\_mdrv\_cart\_name([unit-num])**

Returns a status, and the name of the Microdrive cartridge in *unit-num* (default of 1). The returned cartridge name is only meaningful when the status returned is `inks.status.ok`.

```
-- With no Microdrive cartridge inserted into unit 1. Note the status number
print(inks.get_mdrv_cart_name()) → 20    <NO-CART.>

-- With a Microdrive inserted into unit 1
print(inks.get_mdrv_cart_name()) → 0     ChuckieEgg
```

## **inks.get\_model\_name(model [,long\_name])**

Returns the name of the specified mode which should be a value from the `inks.model` table.

```
print(inks.get_model_name(inks.model.ace))      → ACE
print(inks.get_model_name(inks.model.ace, true)) → Jupiter ACE
```

## **inks.get\_num\_frames\_executed()**

Returns the number of machine frames executed in the current session since Inkspector was started.

## **inks.get\_num\_rzx\_rollbacks()**

Returns the number of RZX recording rollback points that are available.

## **inks.get\_num\_tape\_blocks()**

Returns the number of tape blocks contained in the currently inserted tape, or 0 if no tape is inserted.

```
inks.load[[horizons.tap]]
print(inks.get_num_tape_blocks())      →      86
```

## inks.get\_peripheral\_features(peripheral)

Returns a table containing a peripheral's features. At the time of writing, this means 'buttons' such as those on the Multiface devices, or switchable features such as the colour board and character generator on the Chroma 81.

The peripheral is specified as one of the inks.peripheral table values.

```
inks.create_machine("zx81", inks.peripheral.chroma_81)
pf = inks.get_peripheral_features(inks.peripheral.chroma_81)
for k,v in ipairs (pf) do
    print(k,inks.peripheral_feature_name(v))
end
```

Displays

```
1      Character Generator
2      Colour Board
```

## inks.get\_ram\_name(ram-page)

Returns the name of the RAM page specified by an inks.ram value. See [inks.get\\_ram\\_pages\(\)](#) for an example.

## inks.get\_ram\_pages()

Returns a table containing the page number and the sha-1 of its current contents for all RAM pages.

```
pages = inks.get_ram_pages()
for k,v in pairs(pages) do
    print(k,inks.get_ram_name(k),v)
end
```

When run with a 48K Spectrum active, you would see something similar to this:

```
0      bbaf13b894e490cddf16db9bab8aea7274440d0c
5      bfe76dd29139376de973a29ed00d2c211306d31d
2      897256b6709e1a4da9daba92b6bde39ccfccd8c1
```

## inks.get\_recent(item-num[,type])

Returns a path from one of the recent lists. **Item-num** is in the range 0-9 with 0 being the most recently accessed item. **type** is one of the following (specified as a string)

Type	Recent Item
snapshot	Snapshot files
tape	Tape image files
script	Lua script files
mdrv	ZX Microdrive cartridge files
poke	Spectrum Games Database POK files
roll	ZX Printer printer roll files
disk	Floppy disk image files
source	Assembly source files

If **type** is not specified, "snapshot" is assumed. If **item-num** is out of range or **type** is unrecognised, nil is returned.



```
print(inks.get_recent(0))      →      C:\temp\vars.szx
print(inks.get_recent(0, "tape")) →      C:\temp\timing.tap
print(inks.get_recent(1, "tape")) →      C:\temp\ula48.tap
```

## inks.get\_recent\_count([type])

Returns the number of items on the specified recent list (or “snapshot” if type is not specified).

```
print(inks.get_recent_count())      →      10
print(inks.get_recent_count("disk")) →      8
```

## inks.get\_release\_manifest()

Returns a string that identifies the Fossil commit the Inkspector executable was built from.

```
print(inks.get_release_manifest())  →      [93984ed53b]
```

## inks.get\_release\_version()

Returns a string containing the release version.

```
print(inks.get_release_version())   →      2.0.6
```

## inks.get\_rom\_name(rom)

Returns the name of the ROM specified by an inks.rom value. See [inks.get\\_rom\\_pages\(\)](#) for an example.

## inks.get\_rom\_pages()

Returns a table containing the ROM number (from the inks.rom table) and the sha-1 of its current contents for all ROM pages currently present.

```
pages = inks.get_rom_pages()
for k,v in pairs(pages) do
    print(k,inks.get_rom_name(k), v)
end
```

When run with a 48K Spectrum active running the original ROM, you will see this:

```
0      ZX Spectrum 48K      5ea7c2b824672e914525d1d5c419d71b84a426a2
```

## inks.get\_screen\_as\_text([mode])

Returns a table containing the current screen contents as ASCII text. By default the table keys are the line numbers, with the value being the contents of that line.

The optional mode can be 0, 1 or 2 with 0 being the default.

Mode	Behaviour
0	Returns every line from the screen, including empty lines. The keys are the line numbers (e.g. 1-24)
1	Returns only populated (non-empty) lines. The keys are sequential starting from 1, which means you can traverse the table that’s returned using Lua’s ipairs
2	Returns only populated (non-empty) lines. They keys are the line numbers, so they

	may not be sequential if empty lines are present on screen. This means you would have to use Lua's pairs (not ipairs) to traverse the table, which may mean the rows are then shown out of order. See <a href="#">inks.get_basic_program()</a> for more information on the difference between pairs and ipairs.
--	---

Start up a Spectrum 128 the run the following snippet:

```
contents=inks.get_screen_as_text()
tprint(contents)
```

Produces (I've removed most blank lines for brevity)

```
8:      128
9:      Tape Loader
10:     128 BASIC
11:     Calculator
12:     48 BASIC
13:     Tape Tester
14:     _____
23:
24: C 1986 Sinclair Research Ltd
```

Performing the same operation with mode 1

```
contents=inks.get_screen_as_text(1)
tprint(contents)
```

produces the same, but line 23 above (above the copyright messages, which I was remiss in removing manually) has now gone:

```
1:      128
2:      Tape Loader
3:     128 BASIC
4:     Calculator
5:     48 BASIC
6:     Tape Tester
7:     _____
8: C 1986 Sinclair Research Ltd
```

Performing the same operation with mode 2 gives you the screen line numbers as the keys, confirming again that line 23 is empty.

```
contents=inks.get_screen_as_text(2)
tprint(contents)
```

```
8:      128
9:      Tape Loader
10:     128 BASIC
11:     Calculator
12:     48 BASIC
13:     Tape Tester
14:     _____
24: C 1986 Sinclair Research Ltd
```

This function works for all emulated machines.

For the ZX Spectrum models, the CHARS system variable (address 23606, or \$5c36 in hex) is read to locate the character set graphics that are then used to match against the display memory to determine which ASCII character is being displayed. INVERTed characters are matched too. If you want to extract text from a game that uses its own character set, but doesn't set the CHARS system variable, you will need to poke the address of the game's font – 256 into CHARS before calling this function.

For the ZX80 and ZX81 models, the memory pointed to by the D\_FILE system variable is traversed to produce the text.

For the Jupiter ACE it is assumed the screen character map is in ASCII.

## **inks.get\_slt\_levels()**

Returns a status and table containing details of the currently loaded .slt file.

```
inks.load([[c:\temp\myth.slt]])
status, levels=inks.get_slt_levels()
print("status:".status)
tprint(levels)
```

Outputs the following:

```
1:
    1: 1          ← Level number
    2: 33792      ← The decompressed level data size
    3: 4220bb77e84857ec8f4d560d0cc2dd1bec50eca6 ← SHA1 of level data
2:
    1: 2
    2: 33792
    3: ec8c6536cbcc16bac636a7b2c32778da83ecd106
3:
    1: 3
    2: 33792
    3: 13d5926b48cd03e8e2ac47941c44cba27dd58341
4:
    1: 4
    2: 33792
    3: a1206293f7548e8845acad9d3ffdaa3f2db2d259
5:
    1: 5
    2: 33792
    3: 5e9b37003b160554be5b37c99348345aa23c3627
```

## **inks.get\_snapshots\_folder()**

Return the name of Inkspector's sample snapshots folder.

```
print(inks.get_snapshots_folder())    → C:\Program Files\Inkland\InkSpector
2\Snapshots
```

## **inks.get\_speed()**

Returns the current configured emulation speed as a percentage.

```
print(inks.get_speed())    →    100
```

## inks.get\_system\_editor\_cursor\_mode()

Returns the current system editor cursor mode letter. If the system doesn't have a cursor mode system (e.g. Spectrum 128 and later BASIC editors, the Jupiter ACE), nil is returned.

```
print(inks.get_system_editor_cursor_mode()) → K
```

## inks.get\_total\_tstates()

Returns the total number of t-states executed by the current machine.

```
print(inks.get_total_tstates()) → 4839380003
```

## inks.get\_z80\_register(reg-name[,reg-name...])

Return the value of one or more Z80 registers.

```
print(inks.get_z80_register("hl")) → 23736
hl = inks.get_z80_register("hl")
print(inks.get_z80_register("bc", "de")) → 5911 23737
bc,de = inks.get_z80_register("bc", "de")
```

## inks.has\_disk\_controller()

Returns true if the machine has a floppy disk controller, otherwise false. Currently this returns false for everything except a +3.

## inks.hex(number[,num-bits])

## inks.hex({numbers}[,num-bits])

## inks.hex(string)

Returns **number** as a hexadecimal string, optionally specifying the number of hex digits (as number of bits) required.

```
print(inks.hex(3735928559)) → $DEADBEEF
print(inks.hex(127)) → $7F
print(inks.hex(127,16)) → $007F
print(inks.hex(127,32)) → $0000007F
print(inks.hex("HELLO")) → 48454C4C4F
```

More than one number may be specified by passing in a table containing numbers to convert:

```
print(inks.hex{32,64,128,256}) → $20 $40 $80 $0100
print(inks.hex({32,64,128,256},16)) → $0020 $0040 $0080 $0100
print(inks.hex{inks.get_z80_register("de","hl")}) → $5CB9 $FFFF
```

## inks.insert\_disk(drive-num, path)

Inserts a disk image directly into the floppy drive *drive-num* without invoking any auto-load functionality. A status code is returned.

```
result = inks.insert_disk(0, [[c:\temp\lovely-games.dsk]])
print(inks.status_str(result)) → ok
```

## inks.insert\_tape(path)

Inserts a tape image directly into the cassette player without invoking any auto-load functionality. A status code is returned.

```
result = inks.insert_tape([[c:\temp\pheenix.tzx]])
print(inks.status_str(result))           →      ok
```

### **inks.insert\_mdrv\_cart(cartridge-path[,read-only[,unit-num]])**

Inserts the Microdrive cartridge image file *cartridge-path* into Microdrive unit #1 (or *unit-num* if specified), returning an `inks.status` value. If *read-only* is specified and true, the Microdrive unit will behave as though the recording tab has been removed.

```
result = inks.insert_mdrv_cart([[demo.mdr]])
result = inks.insert_mdrv_cart([[demo.mdr]],true)    ← mount as read-only
result = inks.insert_mdrv_cart([[demo.mdr]],false,3) ← mount into Microdrive #3
```

### **inks.is\_breakpoint(id)**

Returns true if a breakpoint exists with the given id.

```
brk_id = inks.add_breakpoint{type="mem_read",address=32768}
print(inks.is_breakpoint(brk_id))           → true

print(inks.is_breakpoint(32136753))         → false   (or at least, v. unlikely)
```

### **inks.is\_canvas\_enabled()**

Returns whether the canvas is enabled. See [inks.enable\\_canvas\(\)](#) for more information.

### **inks.is\_debug\_build()**

Returns whether the Inkspector executable is a release build (false) or a debug build (true). Released versions of Inkspector will always return false.

### **inks.is\_directory(path)**

Returns true if the specified path exists as a directory, otherwise false.

```
print(inks.is_directory([[c:\temp]]))       →      true
print(inks.is_directory([[c:\tempo]]))     →      false
```

### **inks.is\_disk\_inserted(drive-num)**

Returns true if there's a disk inserted into the specified *drive-num* (i.e. a disk image has been loaded or created). As with all disk related functions, *drive-num* may be 0, "A" or "A:" to specify drive A:, or 1, "B" or "B:" to specify drive B:. The following are equivalent:

```
print(inks.is_disk_inserted(0))             →      false
print(inks.is_disk_inserted("A"))           →      false
print(inks.is_disk_inserted("A:"))          →      false
```

### **inks.is\_disk\_modified(drive-num)**

Retrurns true if the disk in *drive-num* has been written to since it was loaded. Newly created disks return true until they have been saved to a disk image file.

### **inks.is\_drive\_active(drive-num)**

Retrurns true if disk drive *drive-num* is active, i.e. seeking, reading or writing, but not idle.

## **inks.is\_file(path)**

Returns true if the specified path exists as a file, otherwise false.

```
print(inks.is_file([c:\temp\mayhem.szx]))      →      true
print(inks.is_file([c:\temp\lord_lucans_whereabouts.jpg])) →      false
```

## **inks.is\_keyboard\_assist\_active()**

Returns true if Keyboard Assist is active. i.e. is currently “typing” or is waiting for the BASIC editor to become available so it can proceed to type.

## **inks.is\_mdrv\_cart\_inserted([unit-num])**

Returns true if Microdrive #1 (or the unit specified by *unit-num*) has a cartridge inserted.

```
print(inks.is_mdrv_cart_inserted())    → true
print(inks.is_mdrv_cart_inserted(8))   → false
```

## **inks.is\_mdrv\_cart\_modified([unit-num])**

Returns true if Microdrive #1 (or the unit specified by *unit-num*) has a cartridge inserted that’s been written to since last being saved.

```
print(inks.is_mdrv_cart_modified())    → true
print(inks.is_mdrv_cart_modified(8))   → false
```

## **inks.is\_mdrv\_unit\_present([unit-num])**

Returns true if Microdrive #1 (or the unit specified by *unit-num*) is attached, otherwise false.

```
print(inks.is_mdrv_unit_present())     → true
print(inks.is_mdrv_unit_present(8))    → false
```

## **inks.is\_paused()**

Returns true if the machine is paused, otherwise false.

## **inks.is\_peripheral\_attached(peripheral)**

Returns true if the peripheral is attached to the machine, otherwise false. **peripheral** must be one of the values in the inks.peripheral table.

```
print(inks.is_peripheral_attached(inks.peripheral.kempston_joystick)) → true
```

## **inks.is\_playing\_recording()**

This is an alias for [inks.is\\_playing\\_rzx\(\)](#), for compatibility with some old scripts.

## **inks.is\_playing\_rzx()**

Returns true if the machine is currently playing back an .rzx recording, otherwise false.

## **inks.is\_ready\_to\_spool\_line()**

Returns true if the emulated system is ready to receive a spooled lined, otherwise false.

## **inks.is\_recording\_audio()**

Returns true if audio is currently being recorded, otherwise false.

## **inks.is\_recording\_av()**

Returns true if an mp4 media file is currently being recorded, otherwise false.

## **inks.is\_recording\_gif()**

Returns true if an animated .gif is being recorded, otherwise false.

## **inks.is\_recording\_rzx()**

Returns true if an .rzx file is being recorded, otherwise false.

## **inks.is\_spooling()**

Returns true if the system is currently having keypresses spooled to it, otherwise false.

## **inks.is\_supported\_file\_class(path[,{classes-to-include},{classes-to-exclude}])**

Returns true if **path** has a file extension support by Inkspector. If *classes-to-include* is specified as a table of inks.file\_class values, they are not considered supported. Similarly, if *classes-to-exclude* is specified as a table of inks.file\_class values, the results will be filtered on them.

The following script demonstrates using it to filter out files that are Inkspector snapshot types and then create a .scr file from each. This script is supplied with Inkspector as snapshots\lua\batch\_snapshot\_to\_.scr.lua

```
srcfolder = [[C:\my snapshots folder]]
for inpath in dir(srcfolder) do
  -- We're only interested in loading snapshot types (i.e. not screenshots, tapes,
  etc.)
  if inks.is_supported_file_class(inpath, {inks.file_class.snapshot}) then
    lresult = inks.load(inpath)
    -- If the snapshot loaded OK, save out its display...
    if lresult == inks.status.ok then
      -- Save the display a .scr file
      outpath = inks.replace_extension(inpath, ".scr")
      inks.save(outpath)
    end
  end
end
end
```

## **inks.is\_supported\_file\_type(path[,{types-to-include},{types-to-exclude}])**

Returns true if **path** has a file extension support by Inkspector. If *types-to-include* is specified as a table of inks.file\_type values, they are not considered supported. Similarly, if *types-to-exclude* is specified as a table of inks.file\_type values, the results will be filtered on them.

```
srcfolder = [[C:\my z80 folder]]
for inpath in dir(srcfolder) do
  if inks.is_supported_file_type(inpath, {inks.file_type.z80}) then
    lresult = inks.load(inpath)
    if lresult == inks.status.ok then
      outpath = inks.replace_extension(inpath, ".szx")
    end
  end
end
```

```

        inks.save(outpath)
    end
end
end

```

## inks.is\_system\_error\_code(err-code)

Returns true if err-code is a valid system error code for the current machine.

When run on a 48K Spectrum:

```

print(inks.is_system_error_code("0")) → true      (0 OK)
print(inks.is_system_error_code("D")) → true      (D BREAK - CONT repeats)
print(inks.is_system_error_code("R")) → true      (R Tape Loading Error)
print(inks.is_system_error_code("Z")) → false

```

## inks.is\_system\_ready\_for\_keyboard\_assist()

Returns true if the machine has booted up and is ready to accept input, otherwise false.

Returns true if the machine has booted up and is ready to accept input, otherwise false.

## inks.is\_tape\_present()

Returns true if there is a tape currently loaded into the [cassette player](#), otherwise false.

## inks.joystick\_name(joystick)

Returns the name of the joystick for the given **joystick** value.

```

print(inks.joystick_name(inks.joystick.fuller)) → Fuller Joystick

```

## inks.keyboard\_assist(command)

Queues up **command** to be “typed” out by Keyboard Assist when the BASIC editor next becomes available.

```

inks.keyboard_assist([[PRINT "Ooh, this is a little spooky!"]])

```

## inks.keyboard\_assist\_select\_system\_menu(menu-num)

For 128k Spectrums and later, when they are displaying the top level system menu, this function allows one of the menu items to be selected via Keyboard Assist. **menu-num** should be 0 for the first menu item, 1 for the next, and so on.

An inks.status code is returned indicating whether the function succeeded. If the system doesn’t have a system menu (such as a 48k Spectrum) inks.status.unsupported is returned. If the system does have a system menu, but the menu is not currently active, inks.status.cancelled is returned.

Running the following on a freshly booted Spectrum 128 machine will select the calculator menu:

```

inks.keyboard_assist_select_system_menu(2)

```

## inks.line\_count(text)

Returns the number of lines in the string **text**.

```

print(inks.line_count([
line one
line two

```



```
line three  
]]))
```

→ 3

## **inks.load(path)**

Loads a supported file type, returning an `inks.status` code. All file types may be loaded with this one function - snapshots, tape images, ZX Microdrive cartridge images, disk images, POK files, Lua scripts and assembly source files.

## **inks.load\_as(path, as-extension)**

This barmy function attempts to load the file **path** and process it as though it had the file extension **as-extension**. It exists only for one of the Inkspector test scripts that performs fuzz testing. An `inks.status` code is returned.

## **inks.load\_file\_data(path[, fuzz-data, fuzz-num])**

This (also fairly barmy) function loads the contents of the file **path**, optionally writing random values to parts of it when **fuzz-data** is true. By default, 100 bytes of the file at random positions are overwritten, but this can be changed by specifying **fuzz-num**. If it's a positive value, it's taken as the number of bytes of the read file to scramble. If it's negative it's taken as a percentage value of the loaded file size.

Needless to say, this function (or at least, its fuzz parameters) is really of any use to the fuzzer test scripts run prior to an Inkspector release.

## **inks.load\_from\_library([mark-as-selected[,file-num]])**

Loads a file from the library configured by Options → Library. If **file-num** is not specified, a random file is selected, otherwise the *file-num*-th file in the library is selected. If **mark-as-selected** is specified, it controls whether the selected file is marked as having been selected. i.e. whether it could randomly be selected again (until all files have been selected, at which time all files are automatically made available for selection again).

```
inks.load_from_library()  
inks.load_from_library(true)  
inks.load_from_library(false, 42)
```

This function returns an `inks.status` value.

## **inks.log(msg[,message-type])**

Sends a message to the message window (GUI) or console (CLI), optionally specifying the message type. See [inks.add\\_message\\_for\\_user](#) for the available **message-types**.

```
inks.log("Logging a normal message")  
inks.log("Logging a warning", inks.msg_type.warning)
```

## **inks.may\_spool()**

Returns true if the current machine supports spooling. Currently this function will always return true as all emulated machines support spooling (at one point they didn't, and it's possible a machine may be added in the future that doesn't).

## inks.mdrv\_cat([unit-num])

Produces a directory listing of the Microdrive cartridge inserted in Microdrive *unit-num* (or 1 if not specified). Returns a status code and a table containing details of the contents of the cartridge if the status is inks.status.ok.

The details for each file are stored in a sub-table, with a 1-based file number index, followed by details about the cartridge.

For example, running the following snippet with a Microdrive cartridge inserted into unit #1

```
result, dir=inks.mdrv_cat(1)
if result == inks.status.ok then
    print("There are "..dir.num_files.." files on the cartridge")
    tprint(dir)
end
```

Produces:

There are 4 files on the cartridge

1:

```
load_address: 32179
type: 3
num_sectors: 3
filename: Chuckie0.0
length: 1221
raw_filename: Chuckie0.0
hidden: false
```

2:

```
load_address: 53539
type: 3
num_sectors: 24
filename: Chuckie0.M
length: 11997
raw_filename: Chuckie0.M
hidden: false
```

3:

```
program_length: 285
hidden: false
type: 0
num_sectors: 1
filename: ChuckieEgg
length: 285
raw_filename: ChuckieEgg
autorun_line: 10
```

4:

```
program_length: 38
hidden: false
type: 0
num_sectors: 1
filename: run
length: 38
raw_filename: run
autorun_line: 10
```

num\_files: 4

cat\_kbytes\_available: 112

sectors\_available: 225

sectors\_total: 254

bytes\_total: 130048

cartname: ChuckieEgg

bytes\_available: 115200

## **inks.path\_from\_library([mark-as-selected[,file-num]])**

As `inks.load_from_library([mark-as-selected[,file-num]])` except only the path of the selected file is returned and no file is loaded.

```
print(inks.path_from_library()) → C:\Sinclair\RealStuntExperts3.rzx
```

## **inks.pause(pause)**

If **pause** is true, the machine is paused, otherwise it is unpaused. The previous pause state is returned.

```
was_paused = inks.pause(true)
```

## **inks.peek(address-string | address[,ram-page,rom])**

Returns the byte read from **address** with RAM page **page** paged in. If **page** is not specified, the current paging arrangement is used, just as the Z80 would read it.

```
b = inks.peek(65534)
print(b) → 60
```

```
b = inks.peek(0)
print(b) → 243
```

With the ZX Interface 1 is attached, read address 0 from its ROM

```
b = inks.peek("ZXIF1:0")
print(b) → 225
```

NB if a specific ROM is specified but the matching peripheral is not attached, the byte is read using the current paging arrangement, just as the Z80 would read it, so performing the above without a ZX Interface 1 attached would produce:

```
b = inks.peek("ZXIF1:0")
print(b) → 243
```

## **inks.peek\_a(address-string | address[,ram-page,rom])**

As [inks.peek\(\)](#) with the addition that the address, ram page or rom that the byte was read from are also returned.

```
b,addr,ram,rom = inks.peek_a(65534)
print(b,addr,ram,rom) → 60 65534 0 nil
```

Assuming the ZX Interface 1 is attached:

```
b,addr,ram,rom = inks.peek_a(0)
print(b,addr,ram,rom) → 243 0 nil 0
```

```
b,addr,ram,rom = inks.peek_a("ZXIF1:0")
print(b,addr,ram,rom) → 225 0 nil 5
print(inks.get_full_address_str(addr,ram,rom))→ ZXIF1:$0000
```

## **inks.peripheral\_feature\_name(peripheral-feature)**

Returns the name of the specified peripheral feature as a string.

```
pf = inks.peripheral_feature.colour_board
```

```
print(pf, inks.peripheral_feature_name(pf)) → 2 Colour Board
```

## **inks.peripheral\_name(peripheral)**

Returns the name of the specified peripheral as a string.

```
p = inks.peripheral.kempston_joystick
print(p, inks.peripheral_name(p)) → 0 Kempston Joystick Interface
```

## **inks.poke(address-string | address, value [,ram-page])**

Pokes the value **value** at address **address**, with RAM page **page** paged in. If **page** is not specified, the current paging arrangement is used, just as the Z80 would write to it. The previous value at the specified address is returned.

```
old_val = inks.poke(32768,255)
print(old_val) → 0
```

Assuming a 128K Spectrum is running:

```
print(inks.peek("RAM06:49152")) → 0
inks.poke("RAM06:49152", 255)
print(inks.peek("RAM06:49152")) → 255
inks.poke(49152, 128, inks.ram.page_6)
print(inks.peek("RAM06:49152")) → 128
```

## **inks.poke\_a(address-string | address, value [,ram-page])**

As [inks.poke\(\)](#) with the addition that the address, ram page or rom that the byte was poked to are also returned.

```
old_val,addr,ram,rom = inks.poke_a(49152, 64, inks.ram.page_6)
print(old_val, addr, ram, rom) → 128 49152 6 nil
```

## **inks.query\_peripheral\_feature(peripheral, peripheral-feature)**

Returns true or false depending on whether the peripheral feature is available for the specified peripheral currently attached to the machine, or nil if the peripheral is not attached. If true is returned (i.e. the peripheral is attached and has the specified feature) the second parameter is the feature's state, or nil if it has no state.

```
p = inks.peripheral.specmate
f = inks.peripheral_feature.activation_button
-- Create a 48K Spectrum with a SpecMate attached
inks.create_machine("48k", p)
has_feature, state = inks.query_peripheral_feature(p, f)
print(has_feature, state) → true nil
```

## **inks.query\_peripheral\_feature\_state(peripheral, peripheral-feature)**

```
p = inks.peripheral.chroma_81
f = inks.peripheral_feature.colour_board
-- Create a ZX81 with a Chroma 81 attached
inks.create_machine("zx81", p)
```

```
state = inks.query_peripheral_feature_state(p, f)
print(state)                                →      false (colour board not enabled)
```

## **inks.randomise\_ram()**

Sets the contents of the system RAM to random values, simulating power-on conditions.

## **inks.read\_as(data, extension[,add-to-recent])**

Part of the barmy class of functions as used by test scripts, it attempts to load the supplied data (note, data, not a filename) as though it were from a file with the supplied extension.

This example loads the contents of an aufmonty.rzx file, randomly modifying 15% of the loaded data (the -15 parameter), then attempts to load the data as though the data is from a valid .rzx file.

```
path=[[c:\temp\aufmonty.rzx]]
result,data = inks.load_file_data(path, true, -15)
if result == inks.status.ok then
    result = inks.read_as(data, inks.file_extension(path))
end
```

## **inks.read\_mdrv\_cart\_file(path[,unit-num])**

Attempts to read the data from the file named *path* from the cartridge in Microdrive unit *unit-num*, or 1 if not specified. A status code, the data (as a Lua string) and a count of errors encountered while reading the file from the cartridge (e.g. checksum errors) are returned.

```
inks.insert_mdrv_cart([[c:\mdr\Chuckie Egg (1983)(A & F Software).mdr]])
status, data, numerrors = inks.read_mdrv_cart_file("run")
print(status, numerrors, hex(data))      →      0    0
000A2200F1643DBEB0223233373636223AEF2A226D223B643B22436875636B6965456767220D
```

## **inks.relative\_to\_script(filename)**

Returns a new filename made from the supplied filename relative to the name of the running script's filename (if any). See [inks.dofile\(\)](#) (which uses the same filename logic) for details.

## **inks.replace\_extension(path, new-extension)**

Returns a path composed of the extension of the supplied path replaced with new-extension.

This example converts all .scr files within [c:\my\\_scr\\_folder](#) to .gif files.

```
srcfolder = [[C:\my_scr_folder]]
for inpath in dir(srcfolder) do
    if inks.get_file_class(inpath) == inks.file_class.screenshot then
        lresult = inks.load(inpath)
        if lresult == inks.status.ok then
            outpath = inks.replace_extension(inpath, ".gif")
            inks.save_gif(outpath, true)
        end
    end
end
```

## **inks.reset\_machine([hard-reset])**

Performs a soft-reset (i.e. make the Z80 execute code from address 0) , unless **hard-reset** is passed in with a value of true in which case a hard-reset is performed, where a new instance of the current machine is created, simulating turning off and on the power to the machine.

```
inks.reset_machine()      -- Perform a soft reset
inks.reset_machine(true)  -- Perform a hard reset
```

## **inks.resume\_recording\_rzx([path])**

Resumes RZX recording, optionally using the specified snapshot specified as **path**. An `inks.status` is returned indicating the result.

```
inks.result_rzx_recording()
inks.result_rzx_recording([[c:\temp\snapshot.szx]]) -- Resume the recording
after loading snapshot.szx
```

## **inks.rollback\_rzx\_recording()**

Adds a rollback point to the current RZX recording. An `inks.status` is returned indicating the result.

## **inks.save(path [,add-to-recent])**

Saves the state of the current machine to a snapshot with the given path, returning an `inks.status` value. The type of snapshot saved out is determined by the **path's** extension. If **add-to-recent** is specified with a value of true and the save operation is successful, the **path** will be added to the appropriate recent file list.

## **inks.save\_disk(drive-num)**

Saves any changes to the disk image in the specified *drive-num*, returning an `inks.status` value. If the disk image file hasn't yet been saved to a file, `inks.status.not_found` is returned.

```
result = inks.save_disk(0)
```

## **inks.save\_disk\_as(drive-num, path)**

Saves the disk image in the specified *drive-num* to filename *path*, returning an `inks.status` value.

```
result = inks.save_disk_as(0, [[c:\temp\my_lovely_disk.dsk]])
```

## **inks.save\_gif(path[,include-border[,animated]])**

Saves the machine's display to a single-frame (i.e. static) .gif image file, returning an `inks.status` value indicating the result. If `include-border` is specified and true, the machine's border (if any) is included in the .gif. Specifying `animated` as true causes a two frame animated .gif image to be produced if the machine is currently displaying at least one flashing attribute. This can be useful when saving loading screens such as Monty Mole's.

## **inks.save\_mdrrv\_cart([unit-num])**

Saves any changes to the cartridge image for Microdrive unit 1 (unless *unit-num* is specified to override this), returning an `inks.status` value.

```
result = inks.save_mdrrv_cart()
```

## **inks.save\_mdrv\_cart\_as(path[,unit-num])**

Saves the cartridge image for Microdrive unit 1 (unless *unit-num* is specified to override this) to a file called *path*, returning an inks.status value indicating the result.

```
result = inks.save_mdrv_cart_as([[c:\temp\my_lovely_cart.mdr]])
```

## **inks.save\_rzx\_recording(path)**

Saves the in-progress RZX recording to a finalised .rzx file named **path**, returning an inks.status value indicating the result.

## **inks.save\_screen\_as\_text(path)**

Saves the contents of the current machine as ASCII text to the filename *path*.

## **inks.set\_peripheral\_feature\_state()**

Sets a peripheral's feature state, returning an inks.status value indicating the result.

```
p = inks.peripheral.chroma_81
f = inks.peripheral_feature.colour_board
-- Create a ZX81 with a Chroma 81 attached
inks.create_machine("zx81", p)
state = inks.query_peripheral_feature_state(p, f)
print(state)                                →      false  (colour board not enabled)
inks.set_peripheral_feature_state(p, f, true)
state = inks.query_peripheral_feature_state(p, f)
print(state)                                →      true   (colour board enabled)
```

## **inks.set\_profile(controller-type [,profile-name])**

Sets the profile for the given controller-type to **profile-name** or the default profile name ("default") if not specified.

```
-- Set the profile for the keyboard to "default"
inks.set_profile(inks.controller.keyboard)
-- Set the profile for controller #1 to "jetpac"
inks.set_profile(inks.controller.controller_1, "jetpac")

-- Set the profile for the keyboard to one called "mayhem"
inks.set_profile(inks.controller.keyboard, "mayhem")
```

## **inks.set\_profile\_joystick(controller-type, emulated-joystick [,profile-name])**

Sets the controller's emulated joystick type for the default or specified **profile-name**, returning an inks.status value indicating the result.

```
-- Set controller two to emulate a kempston joystick, using the default profile
inks.set_profile_joystick(inks.controller.controller_2, inks.joystick.kempston)
```

## **inks.set\_speed(speed-percent)**

Sets the machine's emulation speed to a percentage specified by **speed-percent** of the original. The speed actually set is returned, which may have been clamped for very large speed values.

## **inks.set\_z80\_register(reg-name,value[,reg-name,value...])**

Set the value of one or more Z80 registers.

```
inks.set_z80_register("af",1234)           -- Set AF to 1234
inks.set_z80_register("ix",0xdead,"iy",0xbeef) -- IX=$DEAD,IY=$BEEF
print(inks.hex{inks.get_z80_register("ix","iy")}) → $DEAD $BEEF
```

## **inks.sha1(string)**

Returns the SHA-1 digest from the contents of *string*.

```
print(inks.sha1("HELLO")) → c65f99f8c5376adaddc46d5cbcf5762f9e55eb7
```

## **inks.sha1\_from\_memory(address, length)**

Returns the SHA-1 digest from the memory contents of the current machine starting at address **address** for a **length** number of bytes.

```
print(inks.sha1_from_memory(0,16384)) →
5ea7c2b824672e914525d1d5c419d71b84a426a2 (for the standard 48K Spectrum ROM)
```

## **inks.sleep(time-to-sleep-for)**

An alias for `inks.dreamysleepynightiesnoozysnooze(time-to-sleep-for)`

## **inks.spectrum\_screen\_address(x,y)**

Returns the screen address for pixel (x,y) with x in range 0-255, and y 0-191, with 0 being the top line. Returns nil if either coordinate is out of range.

## **inks.spool(path | {spool-lines}[,allow-immediate-commands])**

Spools the contents of an ASCII text file (when **path** is supplied as a file name), or one or more lines of text (when **spool-lines** is supplied as a table of text) to the current machine's keyboard entry system. **allow-immediate-commands** controls whether lines without line numbers (i.e. commands that would be executed immediately by the machine) are allowed or ignored. By default they are allowed (default value of true).

Spooling the contents of a file

```
path = inks.get_snapshots_folder()..[["\spool\zx82_union_flag.txt"]]
print("Spooling "..path)
inks.spool(path)
```

Spooling text directly (not from a file). Note the use of curly braces to indicate we're creating a table of values and not a series of individual string parameters.

```
inks.spool({"10 REM Line 1","20 REM Line 2","run"}) -- 'run' will be executed too
```

```
inks.spool({"10 REM Line 1","20 REM Line 2","run"}, false) -- 'run' will not be executed because immediate commands have been disallowed
```

## **inks.start\_recording\_audio(path)**

Starts recording audio to a .wav file with the filename of **path**. If an audio recording is already underway, it is finalised before the new recording is started. An `inks.status` value indicating the result is returned.



## **inks.start\_recording\_av(path[,audio-bitrate][,video-bitrate],[include-border])**

Starts recording an mp4 AV file to a filename of path. If an AV recording is already underway, it is finalised before the new recording is started. An inks.status value indicating the result is returned.

## **inks.start\_recording\_gif(path)**

Starts recording the display to an animated .gif file with a filename of **path**. If a gif recording is already underway, it is finalised before the new recording is started. An inks.status value indicating the result is returned.

## **inks.start\_recording\_rzx(path)**

Starts recording the display to a animated .gif file with a filename of **path**. If a gif recording is already underway, it is finalised before the new recording is started. An inks.status value indicating the result is returned.

## **inks.status\_str(status-code)**

Returns a description for an inks.status **status-code** as a string.

```
print(inks.status_str(inks.status.ok))           →    OK
print(inks.status_str(inks.status.not_found)) →    Not Found
```

## **inks.still\_here()**

Resets Inkspector's internal dead-script counter. Calling this function periodically when performing some computationally expensive operation will prevent Inkspector from determining the current script has stuck and subsequently taking action to stop it. Most scripts will never need to use this as this internal counter is reset every time an “inks” function is called.

## **inks.stop\_keyboard\_assist()**

Stops any Keyboard Assist operation currently in progress and does nothing if it's already idle, or if spooling is active.

## **inks.stop\_playing\_rzx()**

If RZX playback is in progress, it is stopped and this function returns true, otherwise false.

## **inks.stop\_recording\_audio()**

Stops and finalises any audio recording in progress.

## **inks.stop\_recording\_av()**

Stops and finalises any AV recording in progress.

## **inks.stop\_recording\_gif()**

Stops and finalises any animated .gif recording in progress.

## **inks.stop\_recording\_rzx()**

Stops and finalises any .rzx recording in progress.

## **inks.stop\_script([message[,message-style]])**

Stops the currently running script with an optional message with an optional style.

```
inks.stop_script() -- stop the script without any message shown
inks.stop_script("I'm stopping this script")
inks.stop_script("I'm annoyed and stopping this script", inks.msg_type.error)
```

## **inks.stop\_spooling()**

Stops any spooling operation currently underway.

## **inks.system\_error\_str(code)**

Returns the current system's error description, given an error **code**, which may be specified as a number or string.

With a ZX Spectrum running:

<code>print(inks.system_error_str(1))</code>	→	NEXT without FOR
<code>print(inks.system_error_str("1"))</code>	→	NEXT without FOR
<code>print(inks.system_error_str("R"))</code>	→	Tape loading error

## **inks.sysvar(system-variable-name)**

Returns the address and description for a system variable. nil is returned if the variable name is not recognised for the current machine.

<code>print(inks.sysvar("FRAMES"))</code>	→ 23672	3 byte (least significant first), frame counter. Incremented every 20ms
---	---------	---

## **inks.table\_size(table)**

Returns the size of the specified Lua table. Unlike Lua's own table size function, it will return the size of associative tables too.

<code>t={}</code>	
<code>t["test"] = 100</code>	
<code>t[1] = 200</code>	
<code>print(#t)</code>	→ 1
<code>print(inks.table_size(t))</code>	→ 2

## **inks.tape\_is\_code\_loader()**

Returns true if the tape currently loaded appears to be a ZX Spectrum tape that is loaded using a LOAD "" CODE command and not the usual LOAD "". Otherwise false is returned.

```
print(inks.tape_is_code_loader([[Pheenix (1983)(Megadodo).tZX]])) → true
```

## **inks.tape\_play\_from\_block(block-num)**

Causes the currently loaded tape to be played from block number **block-num**, returning true if the operation succeeds.

```
print(inks.tape_play_from_block(10))
```

## inks.temp\_folder()

Returns the location of the temporary folder for the current user.

```
print(inks.temp_folder()) -> C:\Users\Frank\AppData\Local\Temp\
```

## inks.throttle([throttle])

If no parameter is specified, or **throttle** is true, the emulation of the machine is throttled to the currently configured emulation speed. The previous throttle state is returned.

```
was_throttled = inks.throttle(true)    -- Throttle the emulation speed
```

## inks.to\_base(number [,base[,number-of-bits-to-display]])

Returns **number** converted to the specified base, or the current Inkspector hexadecimal or decimal state if no base is specified, and with a minimum number of bits (for binary and hexadecimal).

```
print(inks.to_base(1234))           →    $04D2
print(inks.to_base(1234,2))         →    0b10011010010
print(inks.to_base(1234,2,20))      →    0b00000000010011010010
```

## inks.tokenise\_into\_basic(BASIC-command)

Returns a result code and a BASIC tokenized string generated from the **BASIC-command** string, also expanding abbreviated BASIC commands in the process. This function is used in Key Assist test scripts.

```
result, tokens = inks.tokenise_into_basic("PR. 1")
for n=1, #tokens do
    print(tokens:byte(n))
end
```

Produces the following when run on a 48K Spectrum:

```
245      ← token for "PRINT"
49        ← token for "1"
```

# Inkspector Command Line Tool, incli.exe

The first thing you may ask yourself is “What is the point of a command-line version of Inkspector if I can’t see the emulated machine’s display?”. Good question.

I originally wanted to create a command line tool that would convert from one ZX Spectrum snapshot format to another, handling all permutations of supported snapshots. Want to convert a .snx to a .sp snapshot? No problem! Since the Inkspector code was already split into “core” (i.e. everything emulation related except for playing a sound or displaying a screen) and “GUI” (taking the outputs from the “core”, playing its generated waveforms from the beepers and AY chipsets and showing its emulated displays, etc.) it was easy to knock up a command line tool using the existing “core” code that went something like:

- Create a ZX Spectrum (even though you can’t see or hear its outputs)

- Load a .snx snapshot into the Spectrum
- Ask the Spectrum to save out a .sp snapshots

And of course you could load in any supported snapshot, and save out as any supported snapshot without any additional code, and all the work was already done by the core.

And that was the start of incli.exe. I ended up enhancing it to be able to dump out BASIC listings and variables directly from sources such as snapshot files, tape image files and even Microdrive cartridge image files. And as of the current version, you can even convert .rzx recordings to .mp4 files containing video and audio as easily as typing:

```
incl_i styx.rzx --recav -playrecording
```

Which creates a styx.mp4 file from the styx.rzx recording file.

## Command Line Reference

### --avcfg arg

Configures the audio/video recording parameters as used by the --recav option. The format of *arg* is *parameter=value*. Multiple parameters may be specified, separated by spaces. Supported parameters and values are:

<i>parameter</i>	<i>Value</i>	<i>Effect</i>
profile	baseline	Selects the h.264 baseline profile when encoding video
	main	Selects the h.264 main profile when encoding video
	high	Selects the h.264 high profile when encoding video
aacavgbytesec	AAC average bytes per second.	Sets the average bytes per second value for the AAC (audio) encoder. Supported values are 12000 (default), 16000, 20000 and 24000.
videobps	Video bitrate in bits per second	Sets the target video bitrate for the mp4 file. The default is 800000
border	true or false	Controls whether the machine's border (if it has one) is included in the video recording. The default is false.
destwidth	Width in pixels	Controls the size of the destination video width, allowing it to be scaled. The default is to use the source image width (i.e. not scaled).
destheight	Height in pixel lines	Controls the size of the destination video height, allow it to be scaled. The default is to use the source image height (i.e. not scaled).

Note all the defaults shown above refer to the “factory” defaults, but these may be changed but may be changed on the Inkspector GUI's Options → Recordings page.

For example

```
incli styx.rzx --playrecording --recav --avcfg aacavgbytessec=22050
videobps=1000000 border=false
```

## **--flashgif**

Causes --savegif to preserve any on-screen flashing effect by writing a 2-frame animated GIF file

## **--help**

Shows the help screen, along with the machine names recognised by --machine

## **--hex [arg]**

Sets the global Inkspector number mode to hexadecimal, or forces decimal mode by setting to false. For example, list the system variables for the default machine, first using hexadecimal numbers and then with decimal:

```
incli --listsystem --hex
incli --listsystem --hex=false
```

## **--listbasic**

Displays all BASIC programs stored in any tape image loaded, followed by any programs stored in the first Microdrive unit with a cartridge inserted, followed by any program in memory (probably as a result of loading a snapshot file).

```
incli oink.tap --listbasic

--- START OF TAPE (ZX Spectrum 48k) ---
--- PROGRAM IN BLOCKS 1, 2, FILE 'loader' AUTOSTART LINE 1 ---
1 CLEAR 25855: POKE 23659,0: LOAD ""CODE : FOR f=47105 TO 47117: READ a: POKE
f,a: NEXT f: RANDOMIZE USR 35024: RANDOMIZE USR 0
2 DATA 243,175,211,254,205,142,2,28,40,250,195,69,128
--- END OF TAPE (ZX Spectrum 48k) ---

incli chuckie.mdr --listbasic

--- START OF MICRODRIVE CART 'ChuckieEgg' IN UNIT 1 ---
--- PROGRAM FILE 'ChuckieEgg' AUTOSTART LINE 10 ---
10 CLEAR VAL "24911": BORDER VAL "0": RANDOMIZE USR VAL "23955": LET
a$="Chuckie0.": LET d=PEEK VAL "23766": FOR i=VAL "0" TO VAL "0": LOAD "*"m";d;a$
+STR$ iCODE : RANDOMIZE USR VAL "32179": NEXT i: LOAD "*"m";d;a$+"M"CODE : PAUSE
VAL "001": RANDOMIZE USR VAL "23999"
--- PROGRAM FILE 'run' AUTOSTART LINE 10 ---
10 LET d=PEEK VAL "23766": LOAD "*"m";d;"ChuckieEgg"
--- END OF MICRODRIVE CART 'ChuckieEgg' IN UNIT 1 ---

incli cheqflag.z80 --listbasic
```

```

--- Start ZX Spectrum 48k BASIC In Memory ---
1 BORDER VAL "5": PAPER VAL "5": INK VAL "5": CLEAR VAL "23999": LOAD ""SCREEN$ :
PRINT AT VAL "20",RND;: LOAD ""CODE : RANDOMIZE USR VAL "52646"
--- End ZX Spectrum 48k BASIC In Memory ---

```

## --listbasicvars

Lists the BASIC variables in memory. In the example below, as a result of loading a snapshot into memory.

```

incli oink.szx --listbasicvars

sc=FOR 128 TO 132 STEP 1 @ line 80:2
p=128
s=0
l=0

```

## --listconstants

Lists all the constants provided by Inkspector in the Lua inks table.

## --listdblog

Lists all the messages that have been recorded to the database (see [Logging](#)).

## --listdirectives

Lists all the directives recognised by the Inkspector assembler.

## --listlua

Lists all the functions provided by Inkspector in the Lua inks table.

## --listmdrvcart [arg]

Lists the contents of the cartridge in the first available Microdrive cartridge, or the specified one if *arg* is specified as a value between 1 and 8.

```

incli chuckie.mdr --listmdrvcart

Cartridge name: ChuckieEgg, contains 4 files.
Filename: Chuckie0.0   Type: Bytes   Size:   1221  Autorun LINE: n/a
Load Address: 32179
Filename: Chuckie0.M   Type: Bytes   Size:  11997  Autorun LINE: n/a
Load Address: 53539
Filename: ChuckieEgg   Type: Program Size:    285  Autorun LINE: 10
Load Address: n/a
Filename: run          Type: Program Size:    38   Autorun LINE: 10
Load Address: n/a

```

## --listsysvars [arg]

Lists the system variables for the current machine. If *arg* is specified and set to true, the current values of the system variables are also shown in the display. Use in conjunction with --machine to see the system variables for a different machine.

```
inclcli -listsysvars
```

```
[CLI ] Creating ZX Spectrum 128
SWAP    $5B00    $0014    Swap paging subroutine
YOUNGER $5B14    $0009    Return paging subroutine
ONERR   $5B1D    $0012    Error handler paging subroutine
PIN     $5B2F    $0005    RS232 input pre-routine
etc.
```

```
inclcli -listsysvars=true
```

```
[CLI ] Creating ZX Spectrum 128
SWAP    $5B00    $0014    $FD, $07, $A2, $7A, $65, $76, $F5, $35...      Swap
paging subroutine
YOUNGER $5B14    $0009    $10, $27, $0C, $F6, $DF, $48, $83, $04...      Return
paging subroutine
ONERR   $5B1D    $0012    $9A, $99, $8A, $07, $36, $DF, $CA, $9F...      Error
handler paging subroutine
PIN     $5B2F    $0005    $4A, $EC, $AF, $B5, $E6 RS232 input pre-routine
```

```
inclcli --listsysvars --machine=ace
```

```
[CLI ] Creating Jupiter ACE
FP_WS   $3C00    $0013    19 bytes used as work space for floating point
calculations
LISTS   $3C13    $0005    5 bytes used as work space by LIST and EDIT
RAM     $3C18    $0002    the first address past the last address in RAM
HOLD    $3C1A    $0002    The address of the latest character held in the pad by
formatted output (#, HOLD and so on)
etc.
```

## --listtape

Lists the contents of any tape currently loaded.

```
Inclcli manic.tap -listtap
```

```
The tape image contains 6 blocks
  1: Program "ManicMiner" (line 10) (size 69)
  2: [71 bytes of data]
  3: Code "mmm" (addr 22784) (size 256)
  4: [258 bytes of data]
  5: Code "mm1" (addr 32768) (size 32768)
  6: [32770 bytes of data]
End of tape
```

## --load arg

Loads the supported file *arg*.

```
inclcli zynaps.z80
```

```
[HOST] Loading "C:\temp\zynapsED2.z80"
[Z80R] Z80 version 3 snapshot. Additional bytes: 54
[Z80R] Z80 version 3 snapshot. Hardware: 0, model modifier: 0
[SNPR] Creating machine: ZX Spectrum 48k
[Z80R] Joystick type: 0
[SNPR] Didaktic Melodik is present
[TIMR] Loading "zynaps.z80" took 21.2185ms.
```

## --log

Logs all messages from the session to the file incli\_yyyymmddThh,,ss\_log.txt

```
incl i zynaps.z80
```

```
type incl i_20250623T155129_log.txt
```

```
[HOST] Loading "C:\temp\zynapsED2.z80"
[Z80R] Z80 version 3 snapshot. Additional bytes: 54
[Z80R] Z80 version 3 snapshot. Hardware: 0, model modifier: 0
[SNPR] Creating machine: ZX Spectrum 48k
[Z80R] Joystick type: 0
[SNPR] Didaktic Melodik is present
[TIMR] Loading "zynaps.z80" took 21.2185ms.
```

## --machine arg

Sets the machine to *arg*, where *arg* is one of the names recognised by the assembler's [TARGET](#) directive.

```
incl i --machine plus2a
```

```
[CLI ] Creating ZX Spectrum +2A
```

## --noloadconfig

Prevents the CLI from loading the Inkspector settings as set by the GUI, i.e. it uses factory settings for the session.

## --peripheral arg

Attaches a peripheral to the current machine. Arg should be one of the values listed under inks.peripheral (type incl i --listconstants to see the list).

big_mouth	dk_speech_synth	multiface
boldfield_joystick	eme_soundbox	none
cheetah_specdram	eti_colorvideo	qsboard
cheetah_sweet_talker	fuller_box	specmate
chroma_80	fuller_orator	zebra_joystick
chroma_81	kempston_55_joystick	zon_x81
currah_usource	kempston_joystick	zx_128_ay
currah_uspeech	kempston_mouse	zx_if1
datel_vox_box	lex_van_ay	zx_if2
didaktic_melodik	mikrogen_analogue_joystick	zx_printer
dk_3channel_sound	mikrogen_digital_joystick	

```
incl i --machine 48k --peripheral zx_if1
```



```
[CLI ] Creating ZX Spectrum 48k  
[CLI ] Attaching ZX Interface 1
```

## **--playrecording**

If a .rzx recording file has been loaded, using this option will cause Inkspector to play it back until completion. While it plays back, progress is shown by displaying the time (in emulated time) and how far through the recording it has progressed, as a percentage.

```
inclcli jungle.rzx --playrecording
```

```
[TIMR] Recording playback took 4112.34ms.  
[USER] Recording playback successful: jungle.rzx  
[RZXP] 9939 frames played @ 2418 FPS. Play time: 03m18s. Retriggered INTs: 1
```

## **--playrzxfolder arg**

Ah, now this *is* niche. Given a folder specified by *arg*, inclcli will play every .rzx recording within it (and any subfolders) until completion, using all the cores available on your computer. So if you have a computer with an 8-core processor, 8 .rzx files will be played simultaneously to play through all the .rzx files as quickly as possible. Although even when playing back 8 .rzx files at once, it can take a long time to play through any modest .rzx collection.

I added this command long ago when I was testing changes to .rzx playback and I wanted to check I hadn't broken playback for the majority of .rzx files I have. Unless you want to hear your computer's fan get ready for take-off from having all its cores working flat out, I don't think it will be useful to anyone else.

```
inclcli -playrzxfolder my_rzx_folder
```

You will then see the status updated once a second, showing the progress for each core:

```
1: 65%. 2: 61%. 3: 62%. 4: 30%
```

i.e. how far through each current .rzx playback each core is. Once a core has completed the playback for its current .rzx file, it is given the next one out of the folder to play, until all files have been played back.

## **--playtape**

If a tape has been loaded, inclcli will run the emulation at maximum speed until the tape has paused for more than 7 seconds, which it takes to indicate that the tape loading has completed.

For example, to save the flashing Manic Miner loading screen to a GIF file:

```
inclcli manic.tap --playtape --savegif --flashgif --timeout 25
```

Which tells inclcli to play the tape (manic.tap) for 25 seconds, then take a screen grab, saving it out as a flashing (i.e. 2-frame) .gif file.

## **--postscript arg**

Specifies that the Lua script (i.e. actual script, not a filename) is run after inclcli has finished running the current operation.

## **--prescript arg**

Specifies that the Lua script (i.e. actual script, not a filename) is run before incli starts the requested operation.

## **--quiet**

Suppresses the displaying of general information messages. Errors, warnings and explicit successes are still shown.

## **--readonly**

Instructs incli to work in read-only mode, where files will not be modified or created. This is the same mode that the GUI's file preview uses to avoid inadvertent creation of files. e.g. assembler listings, output binaries, etc.

## **--recaudio arg**

Records audio of the session to a .wav format filename named *arg*.

```
incl_i aufmonty.rzx --recaudio auf.wav --playrecording
```

## **--recav arg**

Records audio and video of the session to an .mp4 file named *arg*. If *arg* is omitted, the name of any loaded file is used, with its extension changed to .mp4.

```
incl_i styx.rzx --recav --playrecording
Generates styx.mp4 from the .rzx recording
```

```
incl_i styx.rzx --recav out.mp4 -playrecording
Generates out.mp4 from the .rzx recording
```

## **--recgif arg**

Records an animated .gif file from the session.

For example, to create an animated .gif of JetPac loading

```
incl_i jetpac.tap --playtape -recgif
```

## **--save arg**

Saves the state of the machine to snapshot file *arg* at the end of the session

For example, to convert the Spectrum snapshot file commando.sna to a .z80 format snapshot:

```
incl_i commando.sna --save commando.z80
```

## **--savegif arg**

Saves the contents of the emulated machine's screen at the end of the session to a .gif file named *arg*. If *arg* is omitted, the name of any loaded file is used, with its extension changed to .gif.

To create jsw.gif from the snapshot

```
incl_i jsw.z80 --savegif
```

## --snippet arg

Runs script snippet *#arg*, where *arg* is between 1 and 10.

```
Wonder what this'll do?  
incl i --snippet 1
```

## --state

Displays the full state of the machine at the end of the session

```
incl i jsw.z80 --state
```

## --timeout arg

Instructs incl i to stop the session after a certain amount of time, even if the operation (such as tape or RZX playback) hasn't completed. If *arg* is a positive value, it is assumed to be seconds. If negative, it's assumed to be the number of emulated frames.

```
incl i manic.tap --playtape --savegif --flashgif --timeout -1250
```

## --verbose [arg]

If *arg* is absent, or true, it enables the display of additional information, equivalent to setting 'Show additional messages that may be useful...' on the GUI's Options → [Advanced](#) page.

# Reference

## Expression Evaluator

### Operators

The available operators are shown below in order of highest priority to lowest.

Operator	Priority	Value Returned
* (unary)	3	The 16-bit value at the given machine address argument (assuming low byte first). For example <code>"*E_LINE"</code> while a Spectrum is running will return the address of the BASIC command being typed in (i.e. <code>PEEK E_LINE + 256 * PEEK (E_LINE + 1)</code> )  To read an 8-bit value at an address use the LOW operator first, e.g. <code>"LOW *E_LINE"</code> .  <i>NB This operator is not available in the assembler as it does not run against a live machine.</i>
~ (unary)	3	The bitwise NOT of the following argument
- (unary)	3	The following argument negated
+ (unary)	3	The following argument (essentially a no-op)
LOW (unary)	3	The low byte of the following argument

HIGH (unary)	3	The high byte of the following argument
WORD (unary)	3	The following argument clamped to a 16-bit value
**	5	<i>Left</i> raised to the power of <i>right</i>
/	5	Divide <i>left</i> by <i>right</i>
*	5	Multiply <i>left</i> by <i>right</i>
%	5	<i>Left</i> modulo <i>right</i>
-	6	<i>Left</i> subtract <i>right</i>
+	6	<i>Left</i> plus <i>right</i>
>>	7	<i>left</i> value shifted right, <i>right</i> number of times
<<	7	<i>Left</i> shifted left, <i>right</i> number of times
>=	8	1 if <i>left</i> is greater or equal to <i>right</i> , otherwise 0
<=	8	1 if <i>left</i> is less or equal to <i>right</i> , otherwise 0
>	8	1 if <i>left</i> is greater than <i>right</i> , otherwise 0
<	8	1 if <i>left</i> is less than <i>right</i> , otherwise 0
!=	9	1 if <i>left</i> is not equal to <i>right</i> , otherwise 0
==	9	1 if <i>left</i> is equal to <i>right</i> , otherwise 0
&	10	Bitwise AND of both arguments
^	11	Bitwise XOR of both arguments
	12	Bitwise OR of both arguments
&&	13	Logical AND of both arguments
	14	Logical OR of both arguments

## Pseudo Variables

Note that these are not available in the assembler, as it does not run against a live machine and has, for example, no Z80 to query.

Pseudo Variable	Value Returned
@m	1 if the Z80's sign flag is set, otherwise 0
@p	1 if the Z80's sign flag is reset, otherwise 0
@z	1 if the Z80's zero flag is set, otherwise 0
@nz	1 if the Z80's zero flag is reset, otherwise 0
@5	1 if bit 5 of the Z80's flags register is set, otherwise 0
@n5	1 if bit 5 of the Z80's flags register is reset, otherwise 0
@h	1 if the Z80's half-carry flag is set, otherwise 0
@nh	1 if the Z80's half-carry flag is reset, otherwise 0
@3	1 if bit 3 of the Z80's flags register is set, otherwise 0

@n3	1 if bit 3 of the Z80's flags register is reset, otherwise 0
@pe	1 if the Z80's parity flag is set, otherwise 0
@po	1 if the Z80's parity flag is reset, otherwise 0
@n	1 if the Z80's subtract flag is set, otherwise 0
@nn	1 if the Z80's subtract flag is reset, otherwise 0
@c	1 if the Z80's carry flag is set, otherwise 0
@nc	1 if the Z80's carry flag is reset, otherwise 0
@rw	The last 8-bit value read from, or written to, memory or port by a Z80 instruction
@addr	The last memory or port address accessed by a Z80 instruction
@ts	The machine's current T-State value
@tapeblock	The current tape block (1 onwards) or 0 if no tape loaded
@tapeplaying	1 if the tape is playing, otherwise 0
@tapepaused	1 if the tape is paused, otherwise 0
@tapestopped	1 if the tape is stopped, otherwise 0
@beamx	The X position of the CRT beam
@beamy	The Y position of the CRT beam
@totalts	The total number of t-states executed since the machine was started up
@zxprinter_state	Returns the internal Inkspector ZX printer stylus state (0-3)
@zxprinter_stylus_pos	Returns the position of the ZX printer stylus, 0-255 if currently over paper, otherwise -1
@zxprinter_rollsize	Returns the length of the current ZX printer paper roll (i.e. number of lines printed so far)
@ram_page	Returns the number of the RAM page currently paged in. Always 0 for systems without paged RAM.
@border	Returns the current border colour (0-7) or -1 for systems that do not display a border.

## Z80 Registers

The value of the Z80's registers are available using the following names:

Register name(s)	Returns
pc, sp, af, a, f, bc, b, c, de, d, e, hl, h, l, af', a', f', bc', b', c', de', d', e', hl', h', l', ix, ixh, ixl, iy, iyh, iyl, i, r, im, wz, iff1, iff2	The value of the register
intvec	The interrupt vector for the current interrupt mode (i.e. the address of the function that will be called when a maskable interrupt is accepted)

affectedflags	1 if the last Z80 instruction executed affected the flags register
halted	1 if the Z80 is currently halted
postei	1 if the Z80 is executing the instruction following an EI instruction
wait	1 if the Z80's /WAIT pin is halting the Z80

If none of the above match the name of the variable, it is then checked against any symbols currently available to the debugger then finally the names of the system variables for the currently running system.

```
print(inks.eval("FLAGS"))      →    $5C3B
print(inks.eval("pc"))         →    0
```

## Source Code Support

### Creating .lst and .sym files using TASM

Squak Valley Software's TASM cross assembler may be used to produce .lst and .sym files compatible with Inkspector by running the following commands:

As an example, using the 16K Spectrum's ROM assembly file "zxsp\_rom\_source.txt" from <https://k1.spdns.de/Vintage/Sinclair/82/Sinclair%20ZX%20Spectrum/ROMs/zx82%20Standard%20Rom/>

and renaming it to zx82.asm, running:

```
tasm -80 -s zx82.asm zx82.obj zx82.lst
```

should produce zx82.lst, zx82.obj (which we can discard) and zx82.sym. The .lst and .sym files can be used on the ROM Management page to provide source code and symbol support in the debugger for the ZX Spectrum 16/48k ROM.

### Creating .lst and .sym files using SjAsmPlus

```
sjasmpus --lst=zx82.lst --sym=zx82.sym zx-spectrum-rom.asm
```

Note Inkspector also supports symbol files produced by its LABELSLIST directive.

### Creating .sym files using PasmO

NB the current version of PasmO does not produce listing files, only symbol files. To create a .sym symbol file compatible with Inkspector, use:

```
pasmO source.asm source.bin source.sym
```

## Supported File Formats

Extension	Description	Support
.sna	ZX Spectrum snapshot format based on the format of memory dumps	R/W

	made by the Mirage Microdriver interface. 48K and 128K versions are supported.	
.snx	48K Spectrum snapshot format	R/W
.z80	Gerton Luntton's ZX Spectrum snapshot format for his Z80 emulator	R (v1,v2,v3) W (v2,v3)
.slt	Damien Burke's Super Level Loader (a .z80 snapshot containing additional level data)	R
.rZX	ZX Spectrum Recording	R/W
.scr	ZX Spectrum screen memory dump. Note Inkspector can also write ZX Spectrum format .scr files when the ZX80, ZX81 or Jupiter ACE are running.	R/W
.mlt	A ZX Spectrum screen memory dump, with the attributes stored per-pixel line, to allow high-resolution effects to be saved accurately.	R/W
.szx	The Spectaculator ZX Spectrum emulator snapshot format	R/W
.rom	ZX Spectrum Interface 2 16K ROM dumps	R
.sp	SPECTRUM / VGASPEC / SPEC386 snapshot formats for 16K and 48K ZX Spectrums	R/W
.tap	Tape container format	R/W
.csW	Compressed Square Wave tape format	R
.tZX	TZX tape format	R/W
.pZX	Patrik Rak's PZX tape format	R/W
.wav	Tape signals stored as a WAVE format file. Mono and stereo PCM and A-law encoded versions are supported.	R
.dsk	Standard and extended disk image format defined by Marco Vieth, Ulrich Doewich and Kevin Thacker.	R/W
.80	ZX80 tape format	R/W
.o	ZX80 tape format	R/W
.81	ZX81 tape format	R/W
.p	ZX81 tape format	R/W
.p81	ZX81 tape format (similar to .p but also contains a filename)	R/W
.ace	Jupiter ACE snapshot	R/W
.mdr	ZX Microdrive cartridge image	R/W
.pok	Spectrum Games Database poke files	R/W
.lua	Lua script	R
.s	Z80 assembly source	R
.asm	Z80 assembly source	R
.zxs	ZX32 Spectrum emulator snapshot format	R
.z81	EightyOne's ZX80 and ZX81 snapshot files	R/W
.zip	All of the above files when archived into a .zip file	R

## Extensions to the .szx format

When Inkspector saves a Spectrum .szx format snapshot and a ZX Printer is currently attached, [if configured to do so](#), it saves out an additional block that contains the contents of the ZX Printer roll using the block below:

```
// Printer roll data is compressed using Zlib
#define ZXSTPPF_COMPRESSED      1

struct ZXSTZXPRINTERPAPER
{
ZXSTBLOCK      blk;           // Standard .szx block header using "ZXPP" as the block id
uint16_t      wFlags;         // ZXSTPPF_COMPRESSED specifies the paper roll data has been compressed
uint16_t      wLinesOfPaper;  // How many lines of paper are stored in chData (0 is a valid value)
uint32_t      dwUncompressedSize; // Size of the uncompressed data
uint8_t      bReserved[4];    // Maybe store printer state in the future

// The compressed or uncompressed paper roll data.
// When loading Zlib compressed data, the compressed size can be obtained by:
// compressedSize = blk.dwSize - ( sizeof( ZXSTZXPRINTERPAPER ) - sizeof( ZXSTBLOCK ) - 1 );
//
// The uncompressed paper roll is stored as a series of 256-pixel lines, using one bit per pixel,
// so 32-bytes per roll line, with a bit of '1' indicating a plot on the paper, starting on the left
// hand side of the paper.
uint8_t      chData[1];       // Start of first line of the roll
};
```

Incidentally, this structure also describes the format of the .zxpri files that may be loaded and saved on the [ZX Printer](#) window.