

Table of Contents

1 Introduction.....	4
1.1 Terms of Use.....	4
2 Getting Started (Showcase).....	5
2.1 Importing a File.....	5
3 Standard Support Tools.....	6
3.1 Unholy Buttons.....	6
3.2 Sprite Pad (SPR).....	7
3.3 Bitmaps (BMP).....	8
3.4 Charsets and Screens (C&S).....	9
3.5 Sine Analysis (SINE).....	10
3.6 Hex Pad (HEX).....	12
4 Lekker Bratwurst.....	13
4.1 Rip Off Koala.....	13
4.2 Sine Analysis (more typical shapes).....	15
8-Bit Samples.....	15
Speed Optimized Code.....	16
4.3 CRAP.....	18
5 The Disassembler.....	19
5.1 The Label Concept.....	19
The good OK, the bad BAD, and the ugly JAM.....	20
5.2 Options and Searchfunctions.....	20
5.3 Searchlists.....	22
5.4 Preview Window.....	24
5.5 NaviMap.....	25
5.6 Pop-Up Menu (Basics).....	26
5.7 Code Shaker and Illegal Shaker.....	27
5.8 Quicksearch and the Quicksearchlist.....	28
5.9 Disable to Data.....	29
6 Showcase.....	30
6.1 Fill the Excluded List.....	30
6.2 Solve the BADs and JAMs.....	30
6.3 Understand the Program Framework.....	32
6.4 Get the IKARI Logo Shaker.....	34
6.5 Get the Logo Flash Routine.....	35
6.6 Get the TSM Y-Movement Routine.....	36
6.7 Get the Scroll Text Flasher.....	37
7 Appendix.....	38
7.1 FAQ.....	38
7.2 Known Bugs.....	38
7.3 "AS IS" Warranty Statement.....	39

List Of Abbreviations (uncompleted)

ASCII	American Standard Code for Information Interchange
BCS	Branch on Carryflag Set
BEQ	Branch on EQual
CIA	Complex Interface Adapter
CPU	Central Processing Unit
CSDb	The C-64 Scene Database
et seq.	and the following
etc.	and so on
IDE	Integrated Development Environment
IRQ	Interrupt Request
FLI	Flexible Line Interpretation
JSR	Jump to SubRoutine
KERNAL	Keyboard Entry Read, Network, And Link
MOS	Metal Oxide Semiconductor
NTSC	National Television Systems Committee
OP-Code	OPeration-Code
PAL	Phase Alternation Line
PEBKAC	Problem Exists Between Keyboard And Chair
PETSCII	Personal Electronic Transactor Standard Code of Information Interchange
RAD	Rapid Application Development
RAM	Random Access Memory
RGB	Red Green Blue
ROM	Read Only Memory
SID	Sound Interface Device
TSM	The Shaolin Monastery
UPX	Ultimate Packer for eXecuteables
VIC II	Video Interface Controller
VICE	Versatile Commodore Emulator

1 Introduction

Once upon a time everyone was eager to find sprites, bitmaps, music or code somewhere in the RAM. Yea, we used the “Action Replay” and other cheat technology to get what we wanted. Infiltrator comes with those basic functionalities including some hopefully nice updates.

The disassembler uses forward interpretation with all the implied problems of this method. The basic concept is to process the complete memory, that’s why you can import PRG files as well as VICE snapshot files. Using VICE snapshots will naturally result in a lot of false interpretations. The included set of tools and methods may help you to master them.

Please note that some support tools were made a long time ago, so they may not have all comfort you know from somewhere else(sprite animations, etc.). The primary purpose of these tools is to help identifying memory areas as graphics, code, etc.

For quick results on your side the manual refers to several programs from different cracking / demo groups using the VICE 2.3 version. Getting these releases in your hands is recommended. Download the latest VICE version here: <http://vice-emu.sourceforge.net/>

Requirements: You should have at least basic knowledge of all MOS Technology chips and a standard computer using the Microsoft Windows XP ServicePack 3 operation system. The software is not tested on any 64 bit operating system yet. You are welcome to try on Vista / Windows 7 and submit any results to me.

Porting requests: The application is programmed in the Lazarus IDE using standard components only. It should be possible to compile the code-lines on various platforms. If you like to volunteer for the job, I will be very pleased. However, give me some time to wait and react on major bugs reported by someone in the first place(plus clean up some source crap).

You are welcome to drop any comment or request to my CSDb mailbox. Search for user RHX / Excess / Secret Lab Productions (SLP)

Cheers,
Gerald

1.1 Terms of Use

Copyright (C) 2008-2011 Gerald Hinder

All rights reserved. This program may be used freely, and you are welcome to redistribute it. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; read the “AS IS” Warranty Statement for details (appendix).

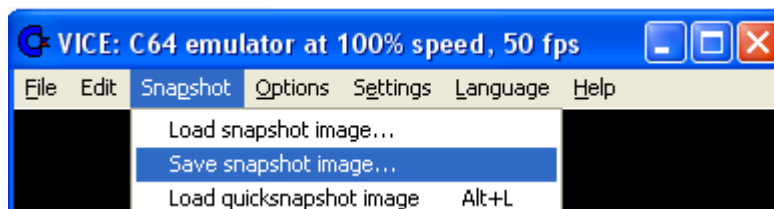
2 Getting Started (Showcase)

For the showcase I opted the “Judge Dredd” crack intro from “IKARI & TALENT + TSM”. Oh no, why this one...?! Because it is less complex which makes it perfect for a showcase. In addition, it features many visual standard techniques you meet in most programs. In case you did not have the release, grab it here: <http://noname.c64.org/csdb/release/?id=17220>.

Once you have loaded the crack using VICE 2.3, the emulator shows you this neat old school intro.

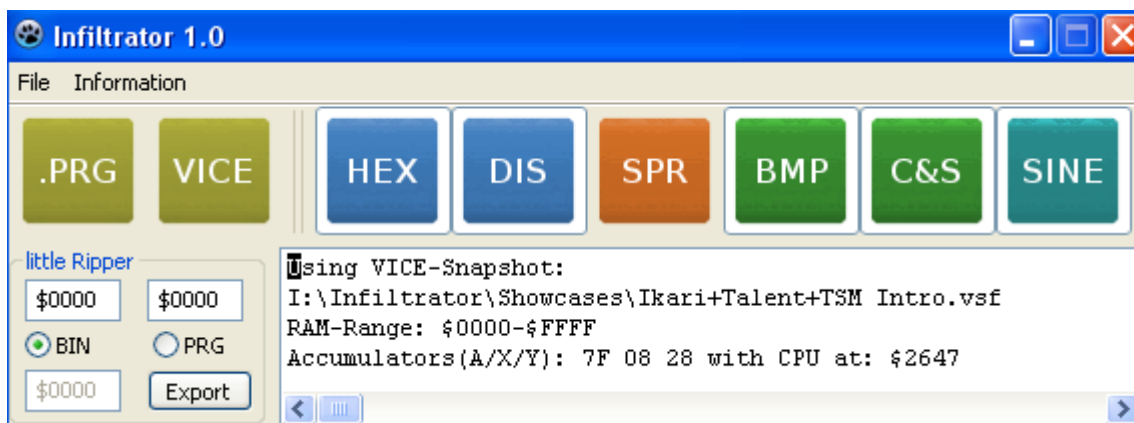


Save a VICE snapshot on your hard disk:



2.1 Importing a File

Simply choose the file type (VICE for the showcase) and select the snapshot you saved before. Depending on the import type, you may get some additional information.



3 Standard Support Tools

Before pushing any buttons, you should always do some preliminary considerations about any program you want to rape. In this case you can expect to find the following:

- the “TSM” sprites (having x and y movements)
- a bitmap font (charset) for the scroll text (“This Game Was ...”)
- the scroll text data
- the charset(s) of IKARI and TALENT logos (probably all in one)
- char data tables of the logos (displayed with some x-sine movement)
- the main screen where everything is displayed
- the music
- some sine for the movements

The main objective is to identify most of these program parts. Later, you can advise the disassembler to ignore parts of the memory, that will reduce errors and false friends.

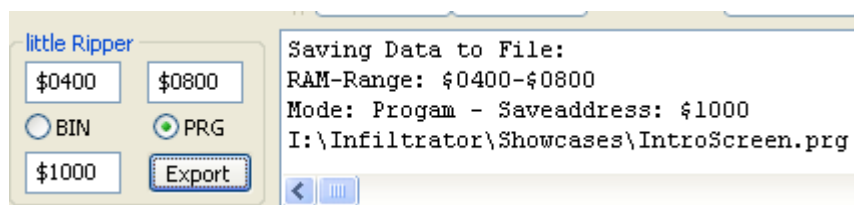
3.1 *Unholy Buttons*

This is for all people who don't care about disassembling, but just want to rip off something. You may already stumbled upon some of these buttons.

- Export as BMP
- Export (little Ripper)
- Make KOALA (see chapter 4)
- Play Data as Wave (see chapter 4)
- Save Data as Wave (see chapter 4)

Export as BMP: Saves the currently shown graphic in a bitmap file (RAD tools, I love you!). You may use Gimp & Adobe products plus Kickass for something bad.

Export (little Ripper): Saves the specified RAM range to a file. Depending on the chosen file type you must enter a valid save address, too. Here is an example how to save the screen data to a new address.



Tip: You can't re-import binary files since the memory allocation is missing. You can still use binary files in your Kickass code.

Tip: Make sure to write the correct suffix in all save dialogs (“.prg”, “.bin”, “.bmp”), I haven't implemented automatism routines for them.

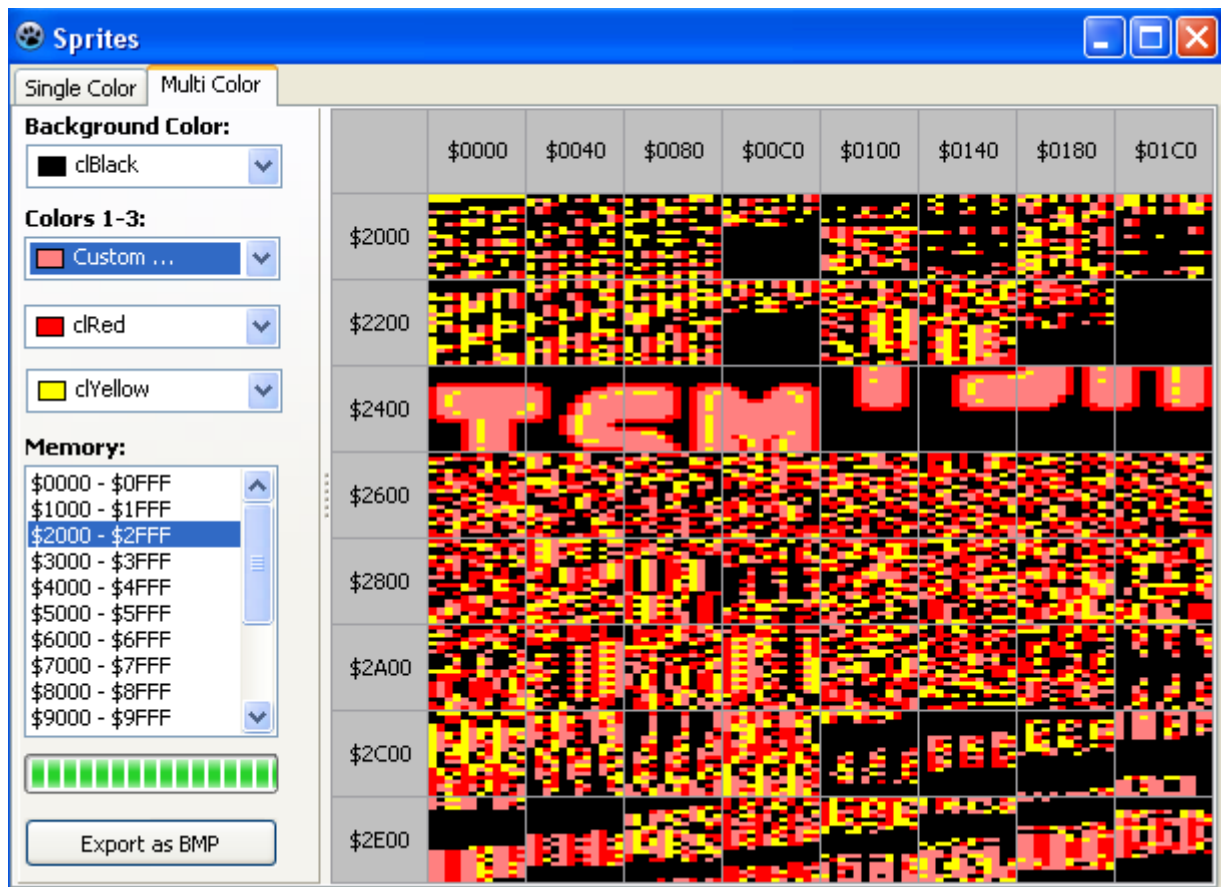
3.2 Sprite Pad (SPR)

Select this button to get there:



Simply choose the memory area you want to display. The multicolored “TSM” sprites can be found at \$2400 - \$25FF and are build out of eight single sprites. You should remember this memory range for the disassembling showcase job(chapter 6).

Tip: The “TSM” sprites memory position(VIC Bank I) indicates that other graphics and the main screen can be found there(\$0000 - \$3FFF).



3.3 Bitmaps (BMP)

Select this button to get there:

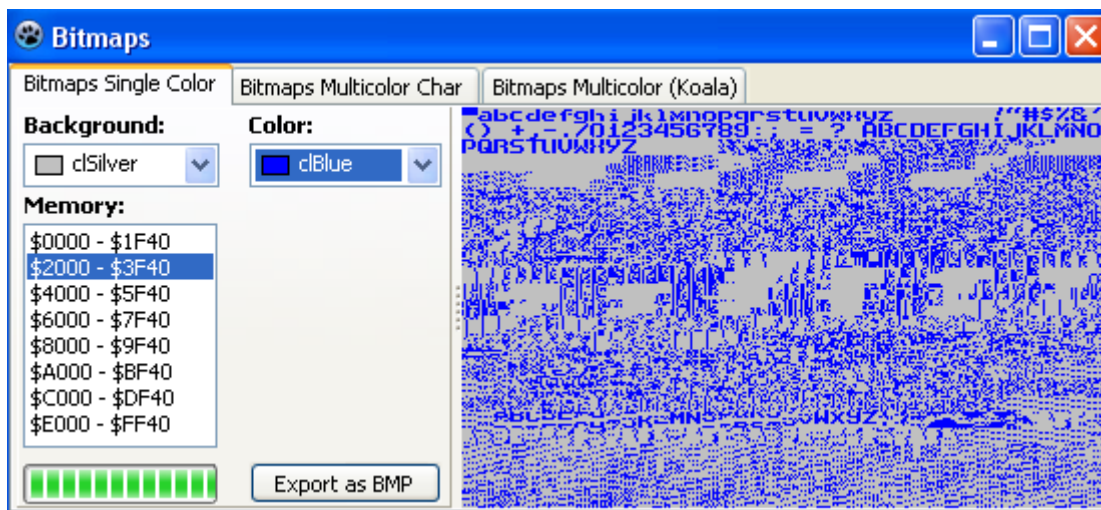
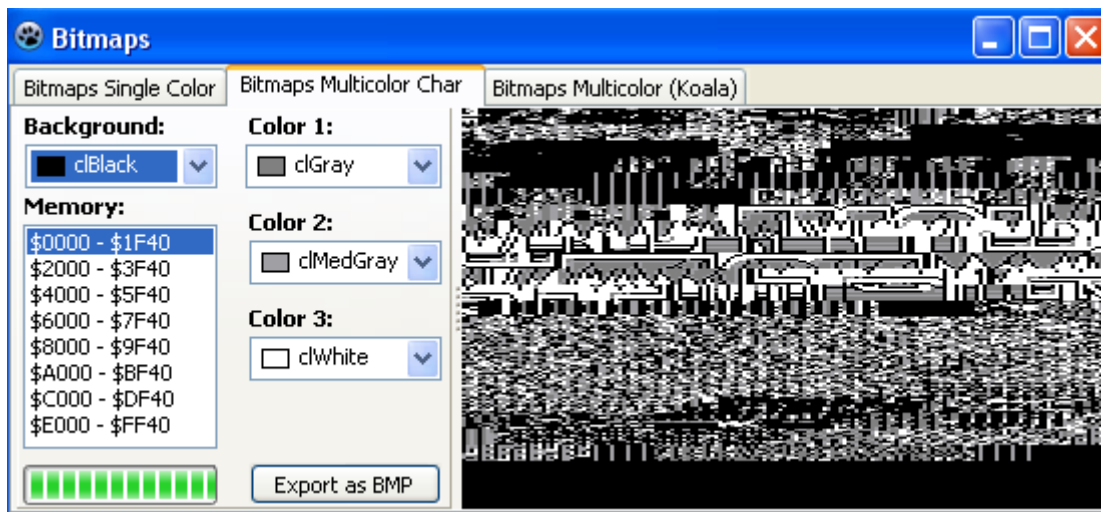


You know the selection procedure, so let's have the results only:

- char based graphic at \$0800 et seq. (exact size unknown yet)
- font bitmap is located at \$2000 et seq. (exact size unknown yet)

=> both has to be verified for the disassembling!

Tip: See chapter 4 for the “Bitmaps Multicolor (Koala)” tab.



3.4 Charsets and Screens (C&S)

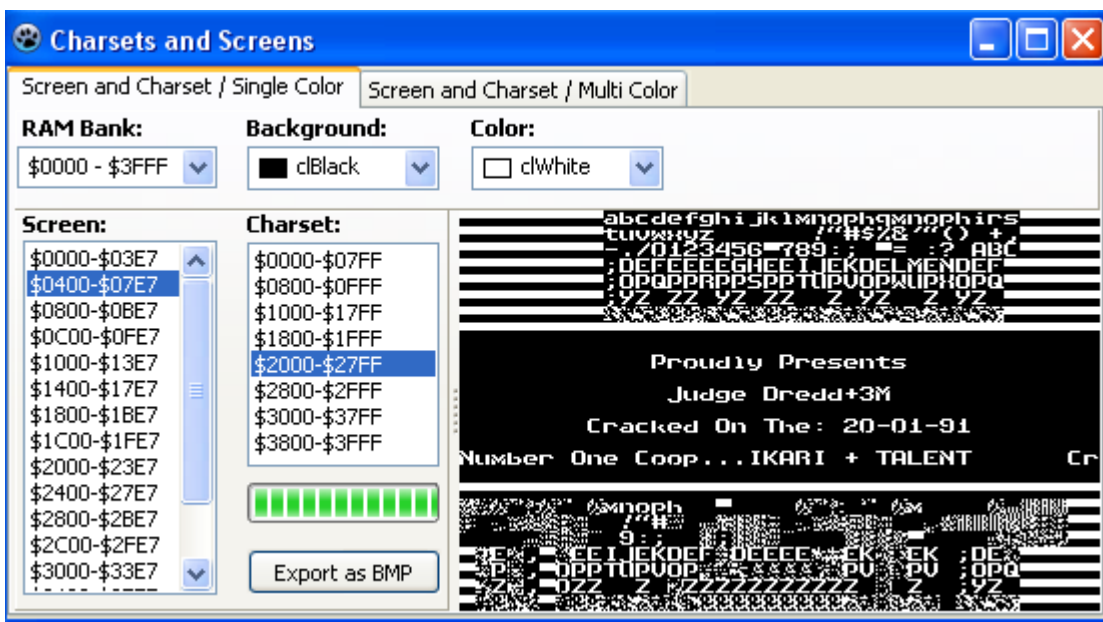
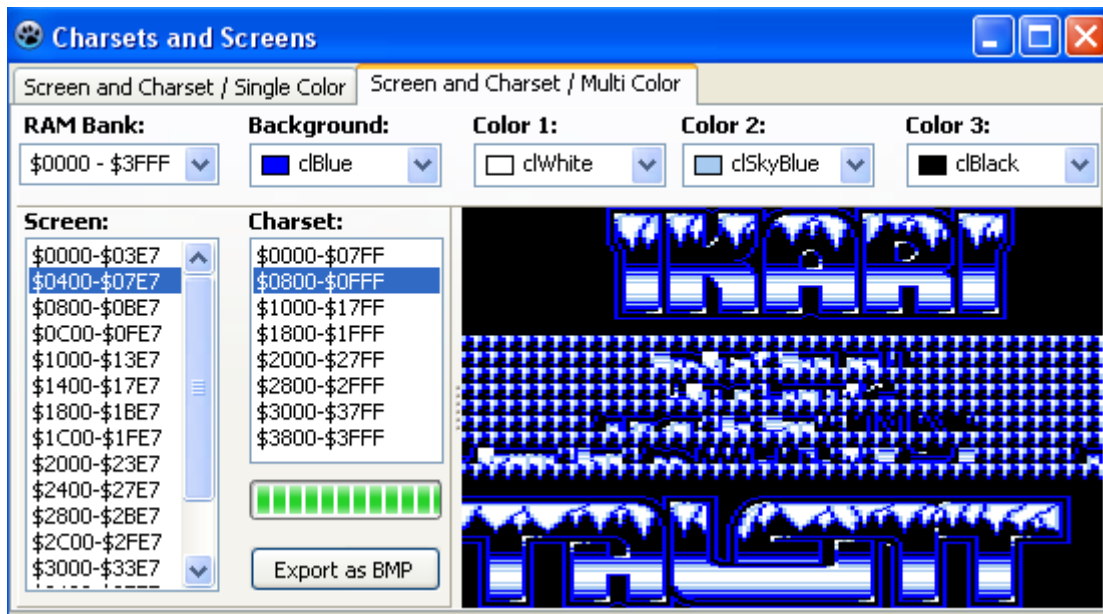
Select this button to get there:



Based on the findings made before it's obvious to search in VIC(RAM) Bank I (\$0000 to \$3FFF). Search for the screen using the identified charsets, results:

- standard screen is used at \$0400
- IKARI and TALENT logos using the same charset

There's no wow function here, so let's continue.



3.5 Sine Analysis (SINE)

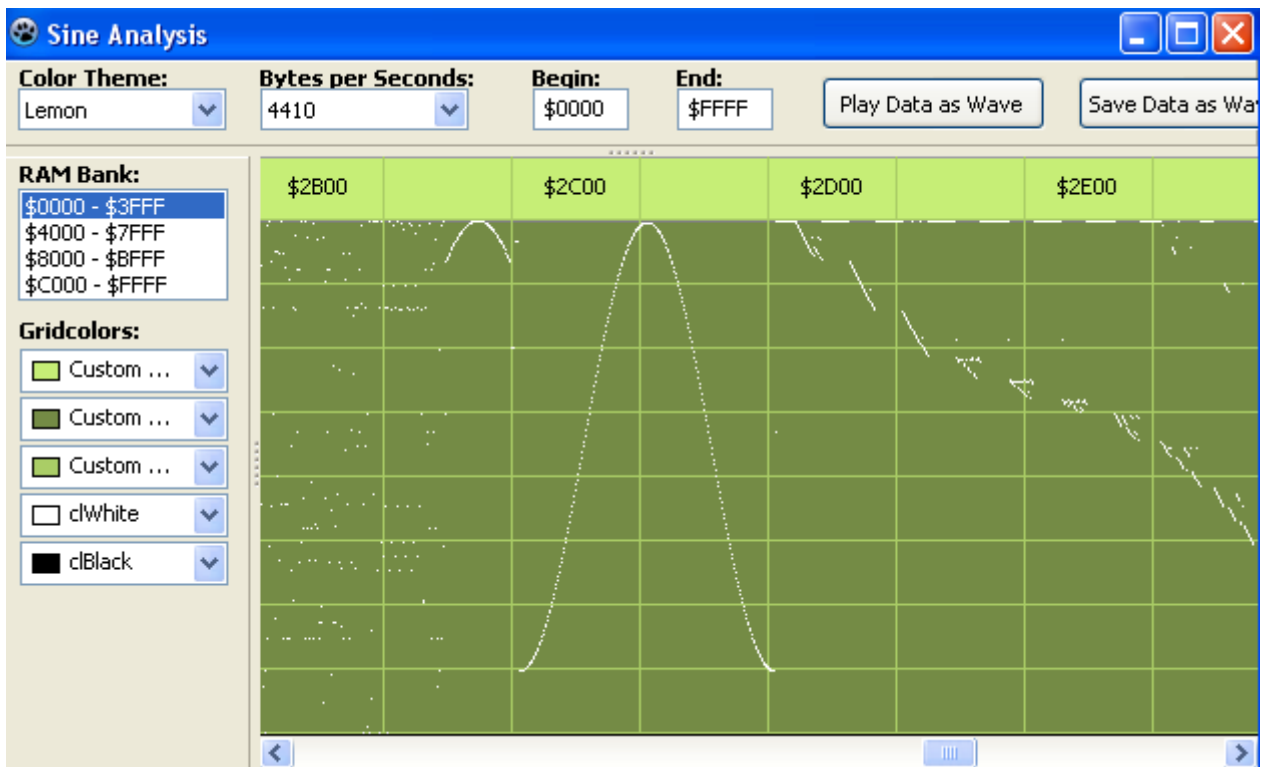
Select this button to get there:



What's this? Well, it displays the RAM values one after the other as pixels in a bitmap. In other words, a neat thing to identify sine waves.

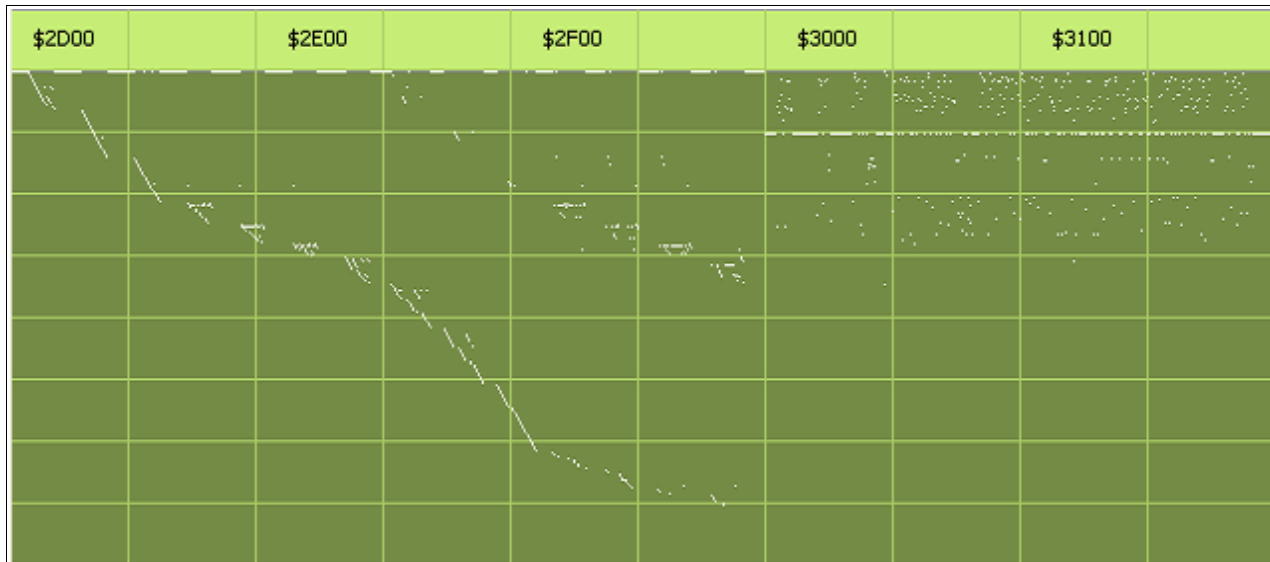
You can easily detect the sine waves used for the logos and sprites. Remember that the TSM sprites moving in x and y direction. Take a **close look**:

- the bouncing half sine uses approx \$40 bytes and starts somewhere at \$2BC0
- the full sine uses approx \$100 bytes but **does not** start exactly at \$2C00

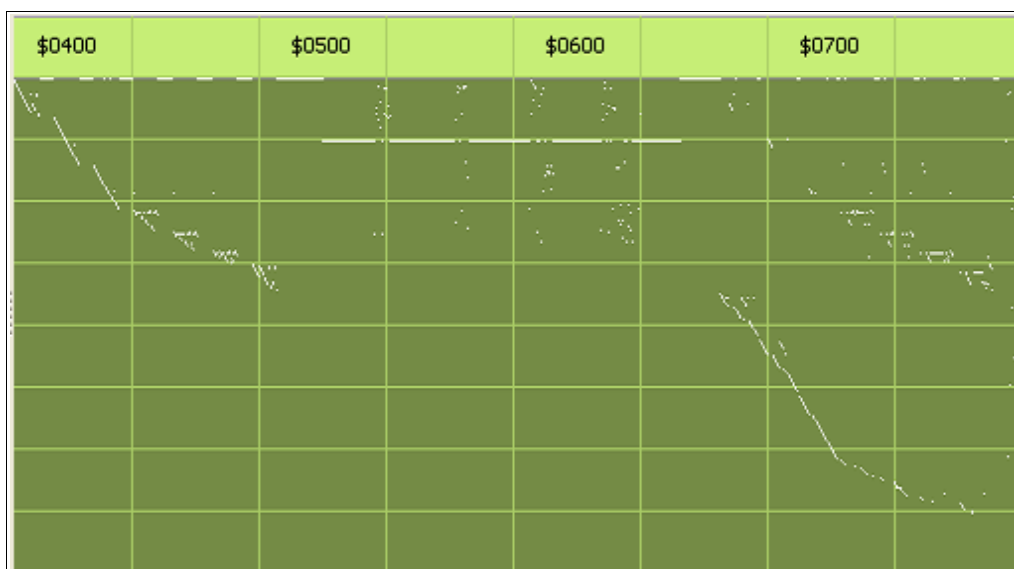


The showcase uses a lot of char based graphics, so let's have a look at it. The memory range from \$2D00 to \$2FFF shows stored char data tables that are used to build IKARI and TALENT logos. The fringed diagonal line indicates to an equal char tool. The standard screen from \$0400 to \$07E7 shows the logos too. But of course, only parts of them!

IKARI and TALENT logos and the scroll text, stored in the RAM:



IKARI and TALENT logos and some text, displayed on the screen:



Text & Scroll-Text:

A small standard font plus the upper letters "A..Z" are used, so you can expect data values from \$01 to \$5F. That's what is shown from \$3000 to \$3200 and from \$0540 to \$06C0. However, the most used char in any text is "Space"(\$20).

3.6 Hex Pad (HEX)

Select this button to get there:



A simple dump of the RAM including a small PETSCII to ASCII conversion. When using VICE as input file, you will have access to additional dumps of some MOS chips(CIAs and VIC-II). Since this is almost standard, anybody knows how it works.

Let's have some findings and updates for the showcase:

- charset from \$0800 to \$0FFF seem to include color tables at \$0F10(logo color flasher?)
- music player plus music data is from \$1000 to \$1A7D
- some comments and blanks from \$1A7E to \$1FFF (can be ignored)
- scroll text use a charset from \$2000 to \$22FF (standard charset plus upper letters)
- bouncing sine from \$2BBE to \$2BFD or \$2BFE
- full sine from \$2C07 to \$2D06 (=> \$2C00 to \$2C06 maybe pointers)
- RAM from \$2D08 to \$2FFF is stored char data for logo shakers
- some text at \$3000 and scroll text from \$3080 to \$3203 (endbyte is \$00, see picture)
- byte \$9E at \$3208 seems to be a basic SYS start command(see picture)

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
\$31C0	2E	2E	20	20	20	20	20	20	20	20	20	20	4C	01	14	05	.. Late
\$31D0	12	2C	20	46	0C	05	14	03	08	20	4F	06	20	49	0B	01	r, Fletch Of Ika
\$31E0	12	09	20	49	0E	20	31	39	39	31	20	20	20	20	20	20	ri In 1991
\$31F0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
\$3200	20	20	20	00	10	08	D0	07	9E	32	30	36	36	20	53	48	@pHg.2066 SH
\$3210	41	52	4B	53	00	A2	00	78	86	01	BD	23	08	9D	F9	00	ARKS@"@.fa=#h].@

That's a all you need know for the showcase disassembling. The following chapter covers the Koala graphics and discuss some typical shapes that may come about in the Sine Analysis tool.

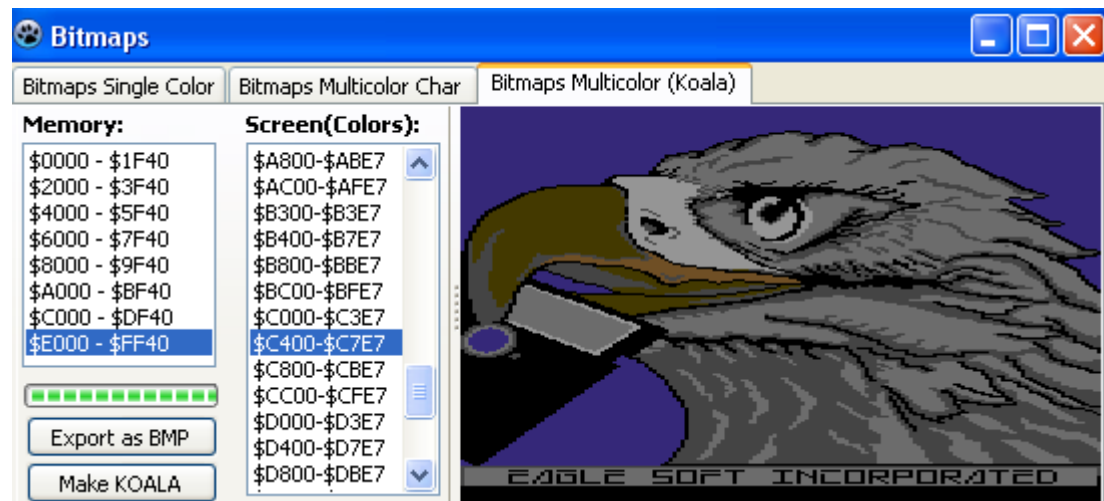
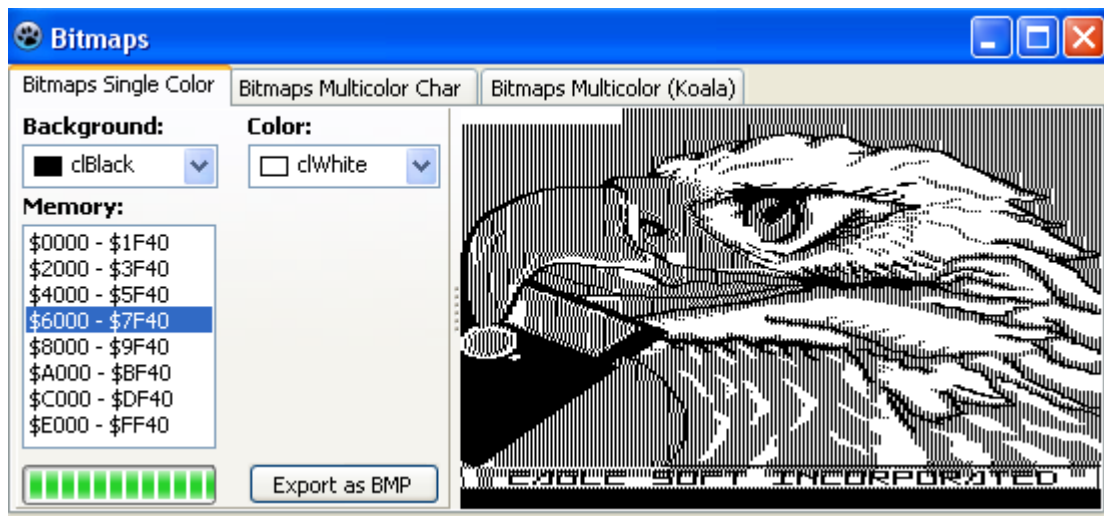
4 Lekker Bratwurst

Guess what's my favourite food. Hmm... this chapter dealing with stuff out of the showcase. Bratwurst rules – and Currwurst too.

4.1 Rip Off Koala

When using VICE you can snatch Koala graphics pretty easy. Do a quick scan on **all** possible memory locations to get the correct bitmap position, switch to Koala mode and search for the screen colors.

But please pay attention, you may come across false friends. Example: The well known “noble bird” is copied by the program for a “fade in” effect from \$6000 in \$E000. You will get stuck by searching for the color table in VIC Bank II only. (Oops... guess what happened to me.)



Because Koala uses screen data to define the pixel colors, an internal palette is used for the recreation and display. An export to a bitmap file will use this palette too, so you can not define that on your own. Here are the RGB values in hexadecimal and integer - in case you need them:

- \$000000 // 0,0,0 (black)

- \$FFFFFF // 255,255,255 (white)
- \$2B3768 // 104,55,43 (red)
- \$B2A470 // 112,164,178 (cyan)
- \$863D6F // 111,61,134 (purple)
- \$438D58 // 88,141,67 (green)
- \$792835 // 53,40,121 (blue)
- \$6FC7B8 // 184,199,111 (yellow)
- \$254F6F // 111,79,37 (orange)
- \$003943 // 67,57,0 (brown)
- \$59679A // 154,103,89 (light red)
- \$444444 // 68,68,68 (dark grey)
- \$6C6C6C // 108,108,108 (grey)
- \$84D29A // 154,210,132 (light green)
- \$B55E6C // 108,94,181 (light blue)
- \$959595 // 149,149,149 (light grey)

The “Make KOALA” button streams the selected RAM to a Koala formatted program file, so you should enter the suffix “.prg”. You can't set the PETSCII code \$C1, it won't be accepted as part of a Windows XP file name. Use the “DirMaster V2/Style” to do that.

In case you are looking for similar robbery routines... nope! Dozens of interlaced and FLI graphic formats cruising around, that's not Infiltrators' assignment yet. There is a nice tool named “Vice Snapshot Grabber 4.2” by Ian Coog/HVSC Crew dealing with this.

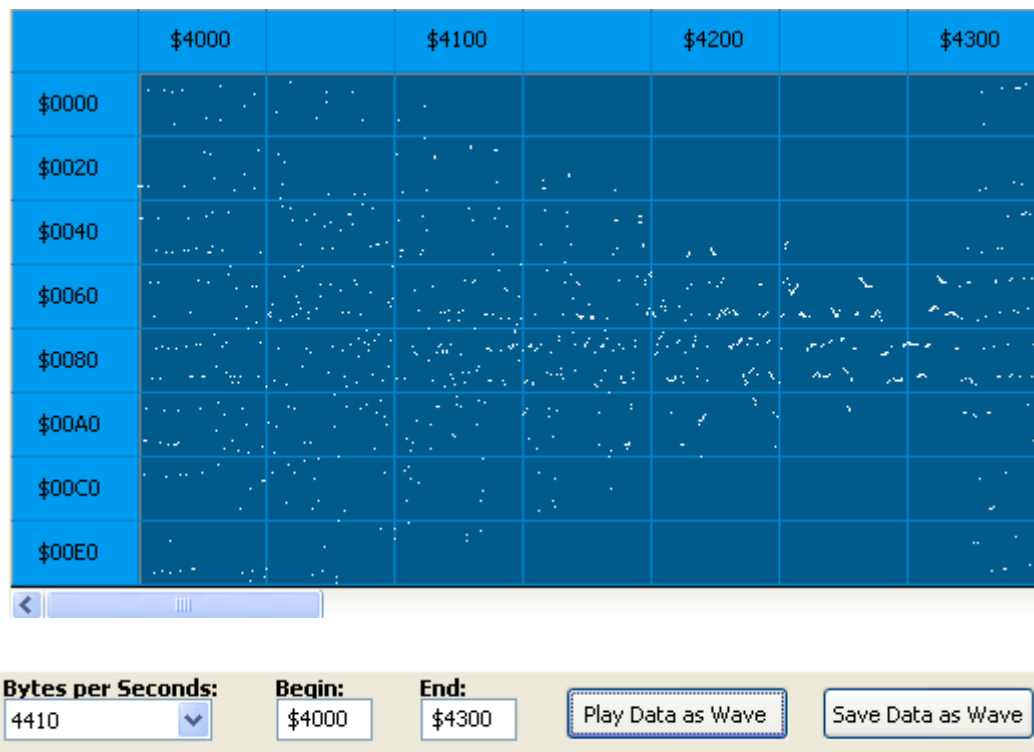
4.2 Sine Analysis (more typical shapes)

8-Bit Samples

Whenever a program uses 8-bit audio samples you get typical audio shapes. I made an internal “data to sound” converter, that's why you can listen to the samples. The little more weird ones can “listen” to graphic, code or whatever else they like. By the way, pay some attention on the BPS value you choose. Playing the complete memory using 441 BPS will take approx 148,6 seconds of unstoppable mythical sound(no thread programming, you know).

The save button creates a Wave formatted file that can be played with any external media player or editor. To make sure it will be saved as “.wav” that signature is always added - whatever you enter as file name.

Here is an example taken from a Megastyle Inc. product:



So, you are actually able to play Cycleburner sounds. Remember that it does not emulate technical effect programming on the SID chip, but just plays the original sample. This feature is rarely tested and I haven't implemented any variations(4-bit, 12-bit) yet.

Fun stuff: In case you got the HVSC collection, you can do this: Make a copy of “I-Ball.sid” (Rob Hubbard) and change the file suffix from “.sid” into “.prg”. Import the PRG file. Due to it's signature “RSID....” it will be loaded from \$5350 to ???? – but that doesn't really matter. Adjust the play beginning and have fun.

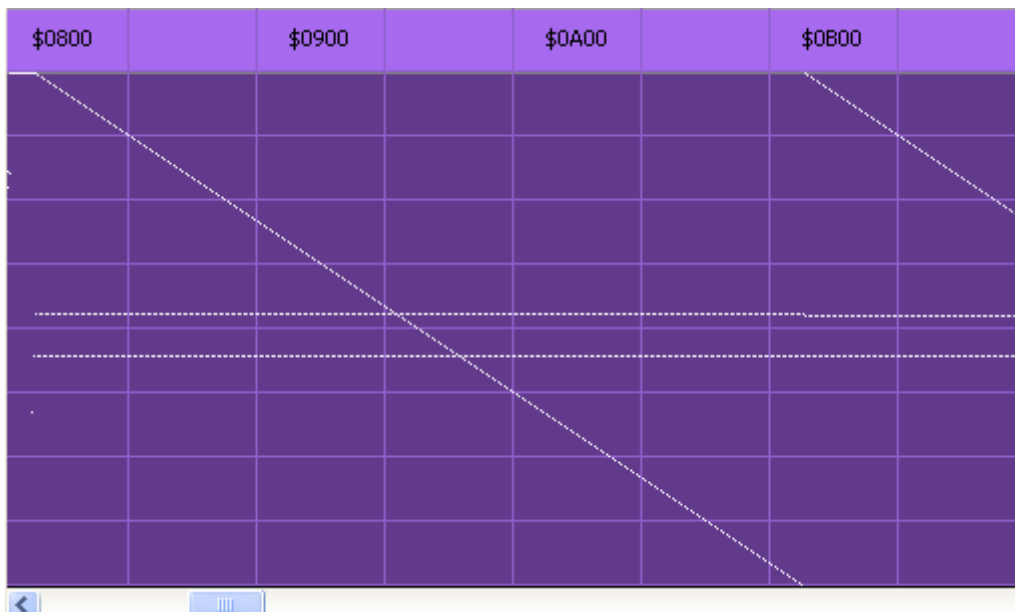
Speed Optimized Code

Code without loops always takes a lot of memory, but is often used in bottleneck situations.

Example 1: Let me start with a code snippet writing accumulator A into RAM \$78XX.

Offset	OP-Code	Low Byte	High Byte	Interpreter
\$0822	8D	00	78	STA \$7800
\$0825	8D	01	78	STA \$7801
\$0828	8D	02	78	STA \$7802
\$082B	8D	03	78	STA \$7803
\$082E	8D	04	78	STA \$7804
\$0831	8D	05	78	STA \$7805
\$0834	8D	06	78	STA \$7806
\$0837	8D	07	78	STA \$7807
\$083A	8D	08	78	STA \$7808

Because OP-Code \$8D and High Byte \$78 are used frequently, both are shown as horizontal, dotted lines. The diagonal line represents the increasing Low Byte.

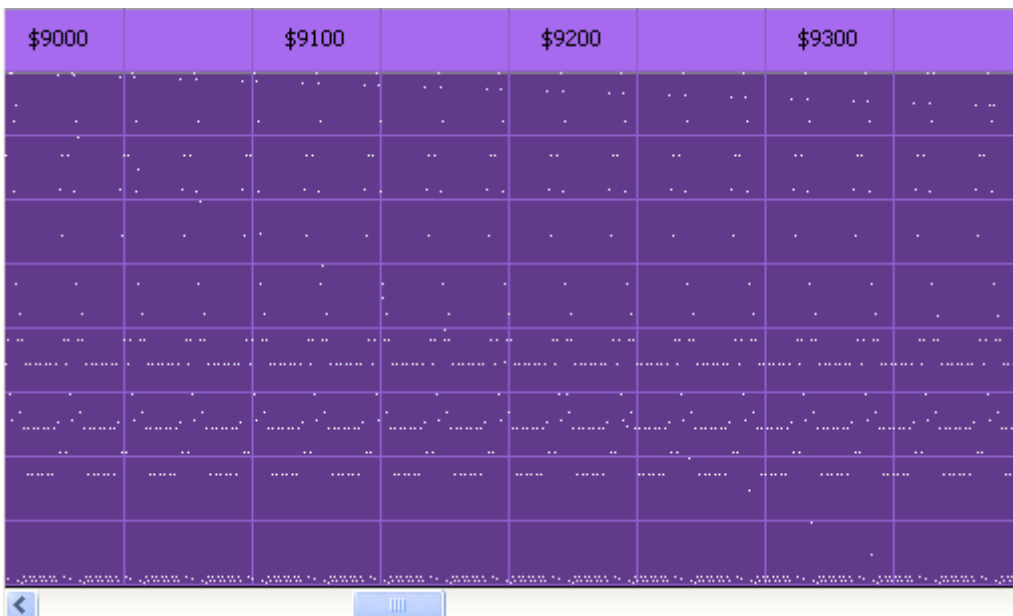


Example 2: The next one shows OP-Codes \$8C and \$A0 wrapping up several VIC II registers at \$D0XX. It's the holy "NU FLI" display routine, © by Crest!

So there's not necessarily a diagonal line, more important are horizontal lines at an OP-Code position.



Example 3: This one uses the indirect addressing mode "LDA (\$FB),Y" and "STA (\$FD),Y".



4.3 CRAP

CRAP allows very little automatism scripting. It's in an early development stage. The following table shows the available commands.

Command	Parameter1	Parameter2	Parameter3	Parameter4
SAVE BINARY	\$Startaddress	\$Endaddress	#filename.bin	-
SAVE PROGRAM	\$Startaddress	\$Endaddress	\$Targetaddress	#filename.prg
MOVE RAM	\$Startaddress	\$Endaddress	\$Targetaddress	-
TRANSFER COLORRAM	\$Startaddress	\$Endaddress	\$Targetaddress	-
TRANSFER CIA1	\$Startaddress	\$Endaddress	\$Targetaddress	-
TRANSFER CIA2	\$Startaddress	\$Endaddress	\$Targetaddress	-
TRANSFER VIC	\$Startaddress	\$Endaddress	\$Targetaddress	-
FILL	\$Startaddress	\$Endaddress	\$Fillbyte	-
INJECT	\$Startaddress	\$Amount of Bytes	\$Bytes, Comma separated	-
UPDATE GRIDS	-	-	-	-

Because some commands modify the internal memory array, your last command should always be UPDATE GRIDS. In case you use the disassembler you should do a re-disassemble for a correct display.

The imported chip data is stored in several arrays, use TRANSFER to access them. The start address of TRANSFER commands should always be \$0000, see examples. This makes only sense when you imported VICE files. Otherwise you get initial values.

For hexadecimal parameters always use **UPPER letters** and don't forget the '\$'. The '#' is used as a parameter signal for file names, so enter it before you type the file name. There is some error handling implemented – but in an early stage. Guess why I called it CRAP.

Here is a senseless example script:

```
SAVE PROGRAM $1000 $1FFF $1000 #audio.prg
FILL $2000 $2FFF $00
TRANSFER COLORRAM $0000 $07E7 $2000
MOVE RAM $0400 $07E3 $2800
TRANSFER CIA1 $0000 $000F $2C00
TRANSFER CIA2 $0000 $000F $2C10
TRANSFER VIC $0000 $002F $2C20
INJECT $2D00 $0C $20,$44,$E5,$A9,$00,$8D,$20,$D0,$8D,$21,$D0,$60
SAVE BINARY $2000 $2DFF #screen_and_chip_rip.bin
UPDATE GRIDS
```

5 The Disassembler

Select this button to get there:



I do not want to bore you too much, but reading this is mandatory.

5.1 The Label Concept

Labels are generated for any direct JMP, JSR and the branches(BNE, BEQ, BCS, etc.). Every label includes the targeting and calling memory address.

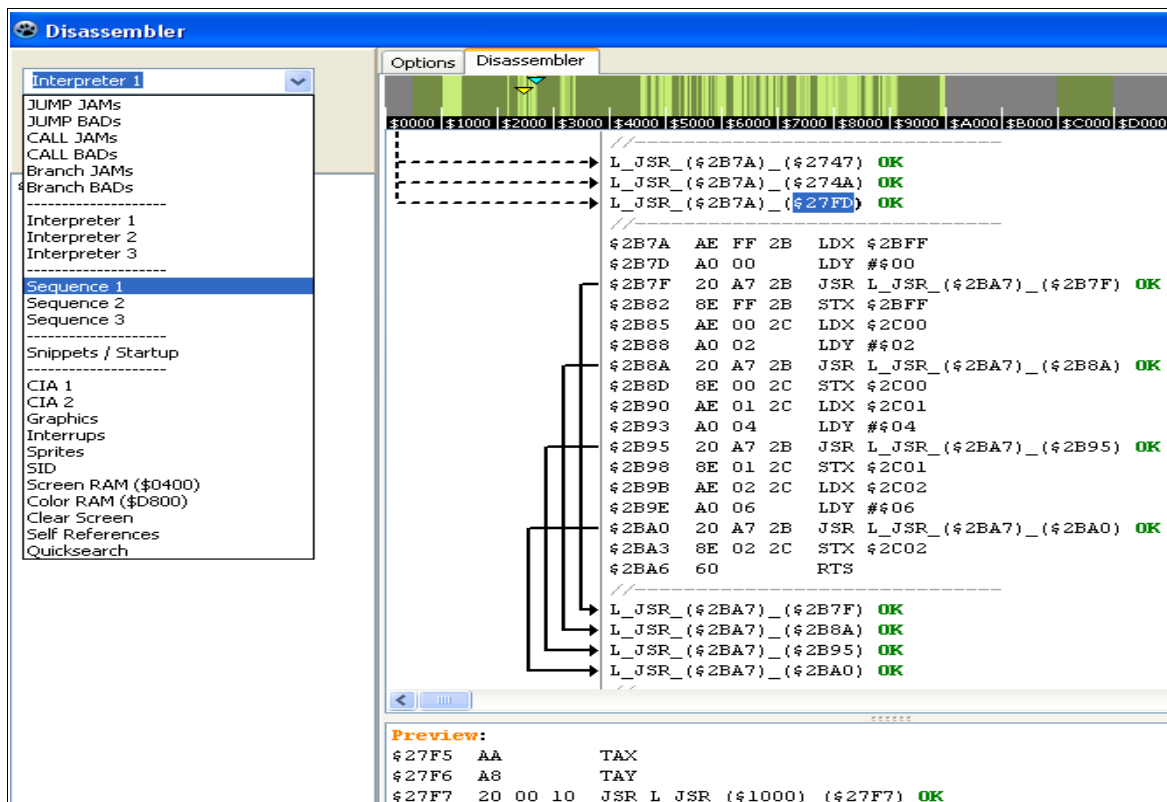
Label definition:

Label-Layout	Values
Signal	L (Label)
Interpreter	JSR, JMP or BRS (Branch)
Target (\$)	(\$XXXX)
Source (\$)	(\$XXXX)

Here is a code line example:

Offset	OP-Code	Low Byte	High Byte	Interpreter	Label	Rating
\$274A	\$20	\$7A	\$2B	JSR	L_JSR_(\$2B7A)_(\$274A)	OK

Here is a look-and-feel:



The good OK, the bad BAD, and the ugly JAM

You may have wondered about the “OK”s above. There is a quality check on every label's destination address. These four results are possible:

Rating	Meaning	Comment
OK	valid OP-Code found	be aware of false friends
BAD	valid OP-Code, but found inside another code line	evidence for crap or incorrect interpretation at target
JAM	invalid OP-Code found	evidence for crap at source and/or target
nothing	target is excluded	

When disassembling, you get BAD and JAM lists for your convenience. I prepared some extra functions to handle them, but first.....options!

5.2 Options and Searchfunctions

Options, quick wins:

Autoadd Comments: Will add clever(?) information to the code.

Colorizer: Gives you some color themes.

CONCAT Bad Bytes: Let them stick together, they may build a row.

Font-/Line Size: Adjust your glasses for free.

Ignore Trap Sequences: Prevents you from let you get thousands of crap results.

Insert Beauty Blanklines: Inserts an extra line after every end of IRQ, RTS, etc.

Progress: Shows the progress of the disassembling.

Start: Start of the program.

Options, Explained Later:

Autobusy: Re-disassemble immediately.

Snatch Snippets: Give me some code fragments.

The Exclude Disassembly Memory List:

Memory ranges to be ignored by the disassembler. Range overlapping is allowed and will not have any effect. Range autosort is activated. You need to re-disassemble after changes were made.

Add to List: Adds an entry to the list.

Delete: Deletes a selected entry, use right mouse button for a pop-up menu.

Save: Save you work to a file(exl = Exclude List).

Load: Import your work done before.

Reset: Give me the initial values.

Linear Interpreter Scan:

Performs a search on the interpreter terms. You can enter a phrase partly, an example: enter “),Y” for all interpreter codes using the indirect Y-indexing addressing mode. Keep in mind that any excluded memory will have an impact on the results.

Byte Sequence Scan:

Performs an old school search that is not affected by any excluded memory.

„only Scan“ Button:

Performs the searches explained above, but don't disassemble.

Options Disassembler

Disassemble
only Scan

Snippets / Start:
 Snatch Snippets
 Start: \$_____

Colorizer:
DisColors
NaviColors

Font-/Line Size
9
2

Autobusy
 Yes
 No

Linear Interpreter Scan:
\$0000 \$FFFF JSR \$1000
\$0000 \$FFFF (\$FB),Y
\$0000 \$FFFF JMP \$08

Byte Sequence Scan:
\$0000 \$FFFF A9,00,8D,20,D0,8D,21,D0,____
\$0000 \$FFFF 78,A9,00,_____
\$0000 \$FFFF 01,58,_____

Exclude Disassembly Memory:
\$0000 \$0000 Little Comment Add to List

```
$0000 - $00FF Zeropage  
$0100 - $03FF extended Zeropage  
$0400 - $07FF Screen  
$A000 - $BFFF Basic-ROM  
$D000 - $DFFF CIA, VIC, SID  
$E000 - $FFFF Kernal-ROM
```

Insert Beauty Blanklines:
 Yes No

CONCAT Bad Bytes
 Yes No

Ignore Trap Sequences:
 JSR \$2020
 JMP \$4C4C

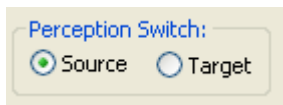
Autoadd Comments:
 Yes No

Progress:

Load Save Reset

5.3 Searchlists

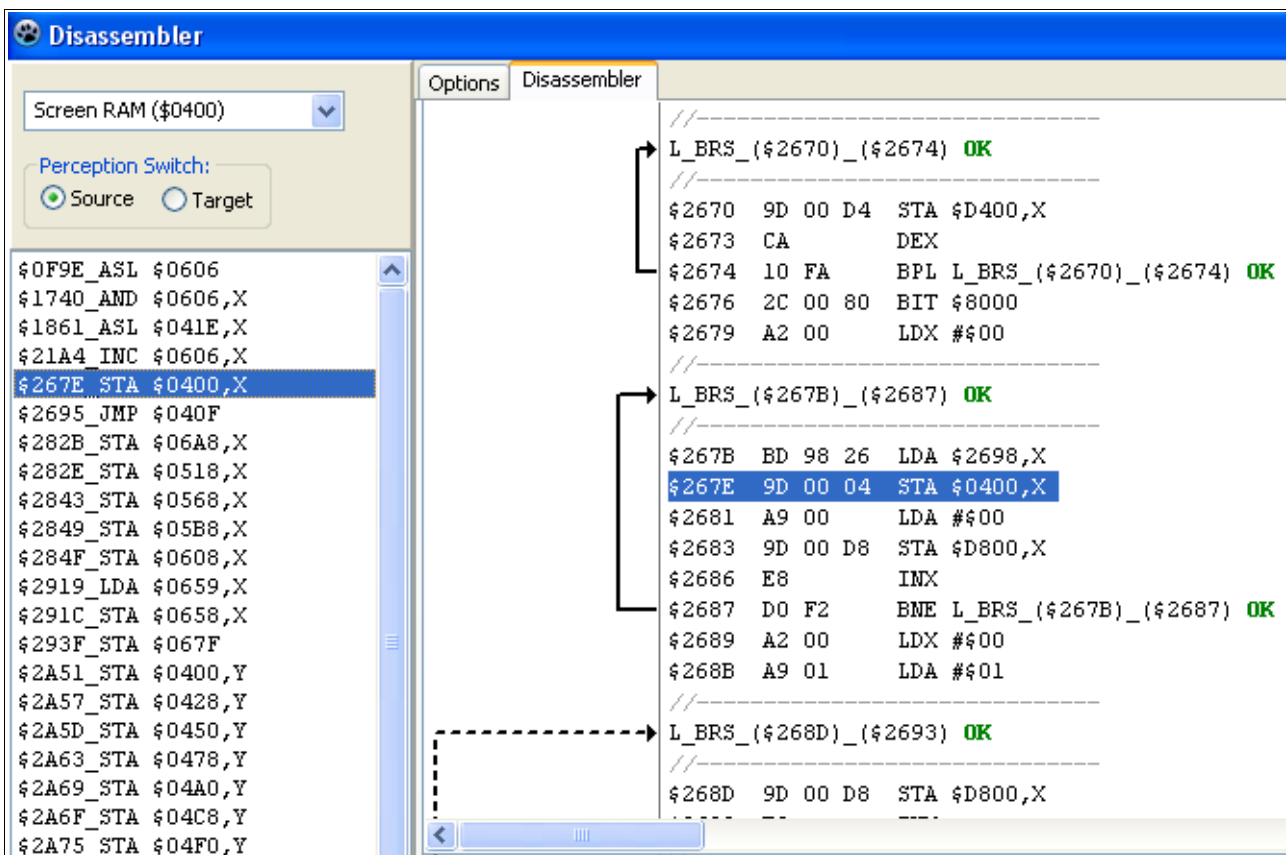
Use the dropdown component to select a generated list and push the left mousebutton to choose an entry. You will be instantly routed to the codeline. Corrupt rated codelines are sorted in six different lists, you can quickly inspect them with this:



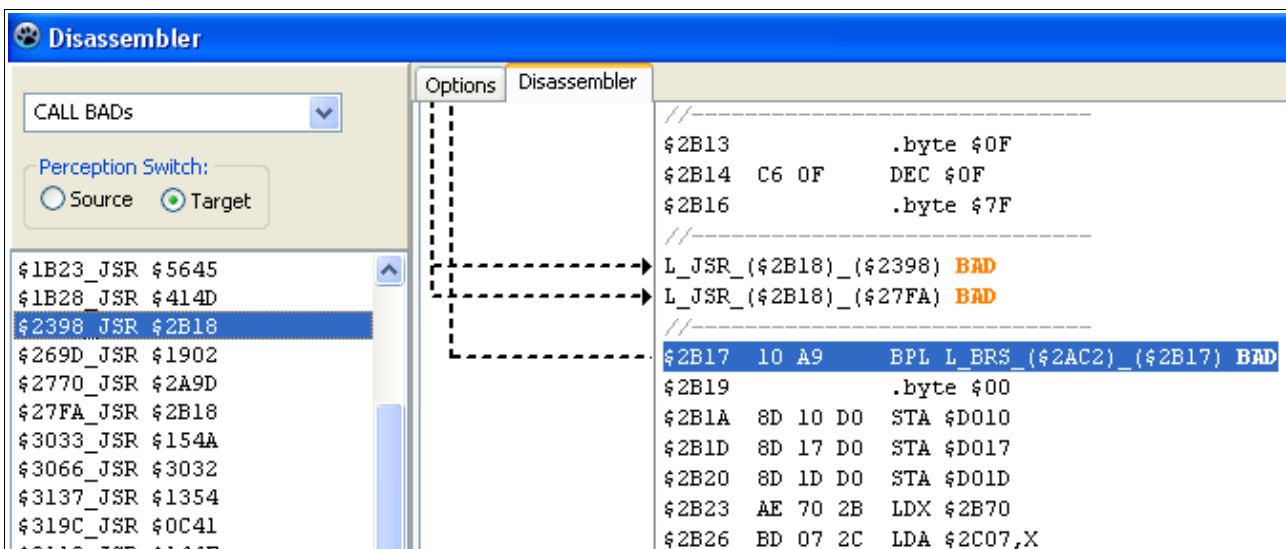
Here is a table with detail information:

Listselection	Description	Perception Switch	Search Type
JUMP JAMs	see above	yes	interpreter
JUMP BADs	see above	yes	interpreter
CALL JAMs	see above	yes	interpreter
CALL BADs	see above	yes	interpreter
Branch JAMs	see above	yes	interpreter
Branch BADs	see above	yes	interpreter
Interpreter 1	see above	no	interpreter
Interpreter 2	see above	no	interpreter
Interpreter 3	see above	no	interpreter
Sequence 1	see above	no	sequence
Sequence 2	see above	no	sequence
Sequence 3	see above	no	sequence
Snippets / Startup	Grabs an unlabeled code line having a preceding line of no code, end of IRQ or RTS. Ten lines of valid code must follow to get an entry. You may find the programstart, code fragments or some IRQ stuff. The given startaddress (options) will show up here, too.	no	interpreter
CIA 1	\$DCXX	no	interpreter
CIA 2	\$DDXX	no	interpreter
Graphics	\$D011, \$D016, \$D018, \$D020, \$D021, \$D022, \$D023, \$DD00	no	interpreter
Interrups	\$D011, \$D012, \$D019, \$D01A, \$0314, \$0315, \$0316, \$0317, \$0318, \$0319, \$FFFE, \$FFFF, \$DC0D, \$DD0D, \$EA31, \$EA7B, \$EA81	no	interpreter
Sprites	\$07F8-\$07FF, \$D000-\$D00F, \$D013-\$D015, \$D01B-\$D01F, \$D025-\$D02E, \$D010, \$D017	no	interpreter
SID	\$D4XX	no	interpreter
Screen RAM (\$0400)	\$0400-\$07E7	no	interpreter
Color RAM (\$D800)	\$D800-\$DBE7	no	interpreter
Clear Screen	JSR \$E544, JSR \$FF81	no	interpreter
Self References	Code target equals code offset	no	interpreter
Quicksearch	Grabs all direct code references to a given address or range. Part of some disassembler popup tools, see chapter 5.8	no	interpreter

Example 1: This is an example how to find anything on the standard screen, so it should not be hard to find scroll or shake routines. The search routines grab any possible OP code combination.



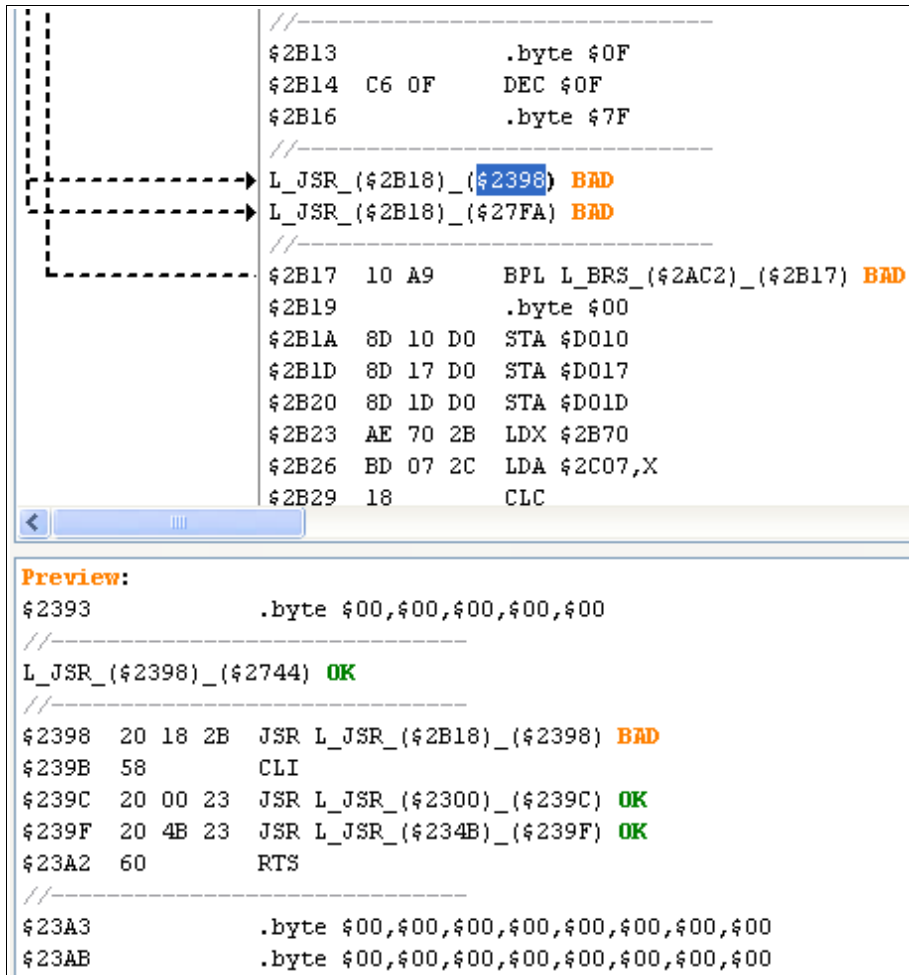
Example 2: This is a typical disassembling error, and a real BAD one too. RAM \$2B18 is called two times(see CALL BADs), additionally the value \$A9 indicates to an LDA #\$00. These are inherent errors triggered by a false interpretation of \$10 at \$2B17. See chapters 5.7 to 5.9 how to analyse and handle this quickly.



5.4 Preview Window

Since you can't use the "Perception Switch" on many lists there is a preview window. A double-click on any valid RAM address will automatically show a preview. It likes every three byte OP-code as well as addresses in labels .

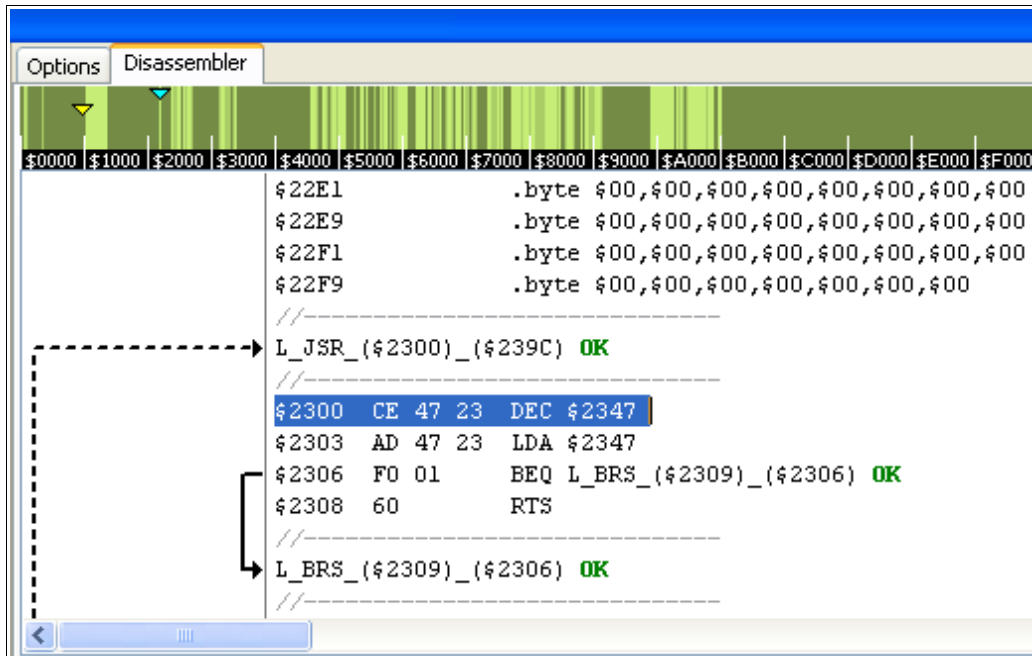
You can quickly stab around if you are curious.



```
//-----  
$2B13      .byte $0F  
$2B14 C6 0F  DEC $0F  
$2B16      .byte $7F  
//-----  
L_JSR_($2B18)_($2398) BAD  
L_JSR_($2B18)_($27FA) BAD  
//-----  
$2B17 10 A9  BPL L_BRS_($2AC2)_($2B17) BAD  
$2B19      .byte $00  
$2B1A 8D 10 D0 STA $D010  
$2B1D 8D 17 D0 STA $D017  
$2B20 8D 1D D0 STA $D01D  
$2B23 AE 70 2B LDX $2B70  
$2B26 BD 07 2C LDA $2C07,X  
$2B29 18      CLC  
  
Preview:  
$2393      .byte $00,$00,$00,$00,$00  
//-----  
L_JSR_($2398)_($2744) OK  
//-----  
$2398 20 18 2B JSR L_JSR_($2B18)_($2398) BAD  
$239B 58      CLI  
$239C 20 00 23 JSR L_JSR_($2300)_($239C) OK  
$239F 20 4B 23 JSR L_JSR_($234B)_($239F) OK  
$23A2 60      RTS  
//-----  
$23A3      .byte $00,$00,$00,$00,$00,$00,$00,$00  
$23AB      .byte $00,$00,$00,$00,$00,$00,$00,$00
```


5.5 NaviMap

This is a concept of weighting the memory. The RAM is portioned in \$80 byte pieces and checked for the code dose inside. The result is a map of the memory that can be used to navigate through the disassembling. A click on the map will route to the selected memory area. The cyan coloured triangle is the current position of your disassembling, while the yellow shows the preview position.



There is the color spread formula.

Colors	Memory Rating
black	unloaded
gray	excluded
dark green	almost crap
green	some code
light green	code

Let's have some application guessing:

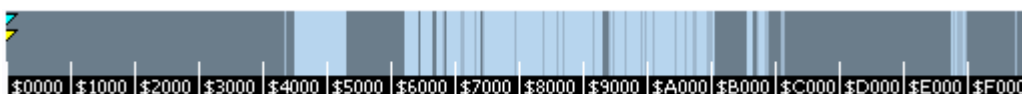
'NUFLI' in VICE with standard excluded areas:



A crunched PRG file, in case you like unpacking routines:

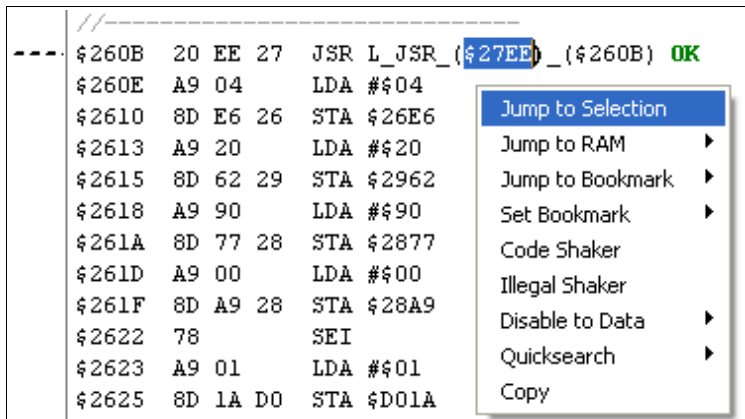


Wizball, in game:



5.6 Pop-Up Menu (Basics)

The disassemble window uses a pop-up menu that gives you access to some more methods.



Jump to Selection: Go to the selected offset address.

Jump to RAM: Let you jump around.

Jump to Bookmark: Go to a bookmarked address.

Set Bookmarks: Set a bookmark for a marked address.

Copy: Copy marked text to the clipboard

The little more complex items “Code Shaker”, “Illegal Shaker”, “Disable to Data” and “Quicksearch” are introduced below.

5.7 Code Shaker and Illegal Shaker

Whenever you are not sure about alternative disassemble interpretations, try “Code Shaker” or “Illegal Shaker” with a marked offset address. Both also like disabled RAM offsets. The results are shown in the preview window. Illegal CPU instructions get an extra marking.

The screenshot displays a disassembler interface with three main sections:

- Top Section:** Shows assembly code with some instructions marked as **BAD**. A context menu is open over the instruction at address `02B17`, with **Code Shaker** selected.
- Middle Section:** A memory dump window showing hexadecimal and ASCII data. Address `03256` is highlighted in blue.
- Bottom Section:** Shows assembly code with instructions at addresses `03256` and `03259` marked as **ILLEGAL**. A context menu is open over `03256`, with **Illegal Shaker** selected.

Code Shaker Section:

```

//-----
//
L_JSR_($2B18)_($2398) BAD
L_JSR_($2B18)_($27FA) BAD
//
02B17 10 A9      BPL L_BRS_($2AC2)_($2B17) BAD
02B19      .byte $00
02B1A 8D 10 D0     STA $D010
02B1D 8D 17 D0     STA $D017
02B20 8D 1D D0     STA $D01D
02B23 AE 70 2B    LDX $2B70
02B26 BD 07 2C    LDA $2C07,X
02B29 18          CLC
02B2A 69 18      ADC #$18
02B2C AA      TAX
02B2D A0 00     LDY #$00
//-----
Shaking: Part one of three...
//-----
02B17 10      .byte $10
02B18 A9 00     LDA #$00
02B1A 8D 10 D0     STA $D010
02B1D 8D 17 D0     STA $D017
02B20 8D 1D D0     STA $D01D
02B23 AE 70 2B    LDX $2B70
02B26 BD 07 2C    LDA $2C07,X
02B29 18          CLC
02B2A 69 18      ADC #$18
02B2C AA      TAX
02B2D A0 00     LDY #$00
//-----
Shaking: Part two of three...
//-----
02B17 10      .byte $10
02B18 A9 00     LDA #$00
02B1A 8D 10 D0     STA $D010
02B1D 8D 17 D0     STA $D017
02B20 8D 1D D0     STA $D01D
02B23 AE 70 2B    LDX $2B70
02B26 BD 07 2C    LDA $2C07,X
02B29 18          CLC
02B2A 69 18      ADC #$18
02B2C AA      TAX
02B2D A0 00     LDY #$00
//-----
Shaking: Part three of three...
//-----
02B17 10      .byte $10
02B18 A9      .byte $A9
02B19 00      .byte $00
02B1A 8D 10 D0     STA $D010
02B1D 8D 17 D0     STA $D017

```

Memory Dump Section:

Address	Hex	ASCII
03226	.byte \$DA,\$51,\$00,\$10,	
0322E	.byte \$DA,\$F0,\$33,\$C9,	
03236	.byte \$FB,\$E6,\$FB,\$D0,	
0323E	.byte \$D0,\$F5,\$E8,\$E6,	
03246	.byte \$FF,\$D0,\$E1,\$A5,	
0324E	.byte \$2D,\$A5,\$FC,\$85,	
03256	.byte \$FF,\$9A,\$A9,\$37,	
0325E	.byte \$4C,\$00,\$20,\$4C,	

Illegal Shaker Section:

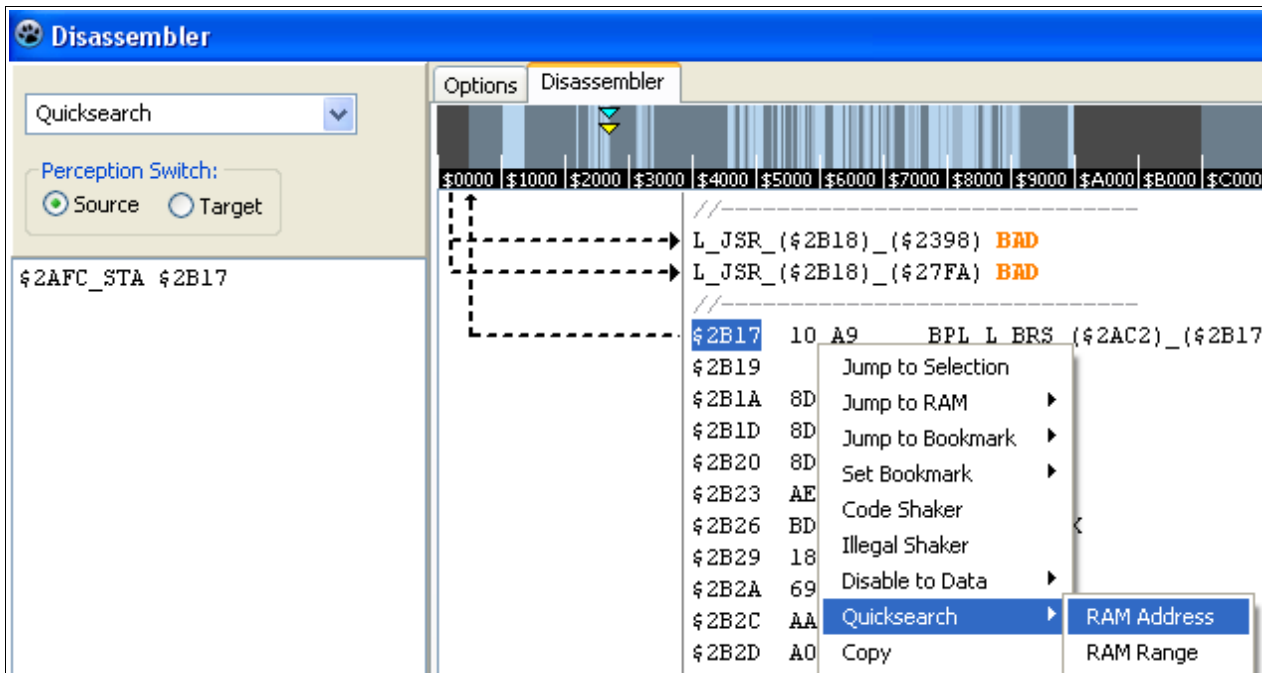
```

//-----
Illegal: Part one of three...
//-----
03256 FF 9A A9    ISB $A99A,X ILLEGAL
03259 37 85      RLA $85,X ILLEGAL
0325B 01 58      ORA ($58,X)
0325D 98        TYA
0325E 4C 00 20    JMP $2000
03261 4C AE A7    JMP $A7AE
03264 E6 FE      INC $FE
03266 D0 02      BNE $326A
03268 E6 FF      INC $FF

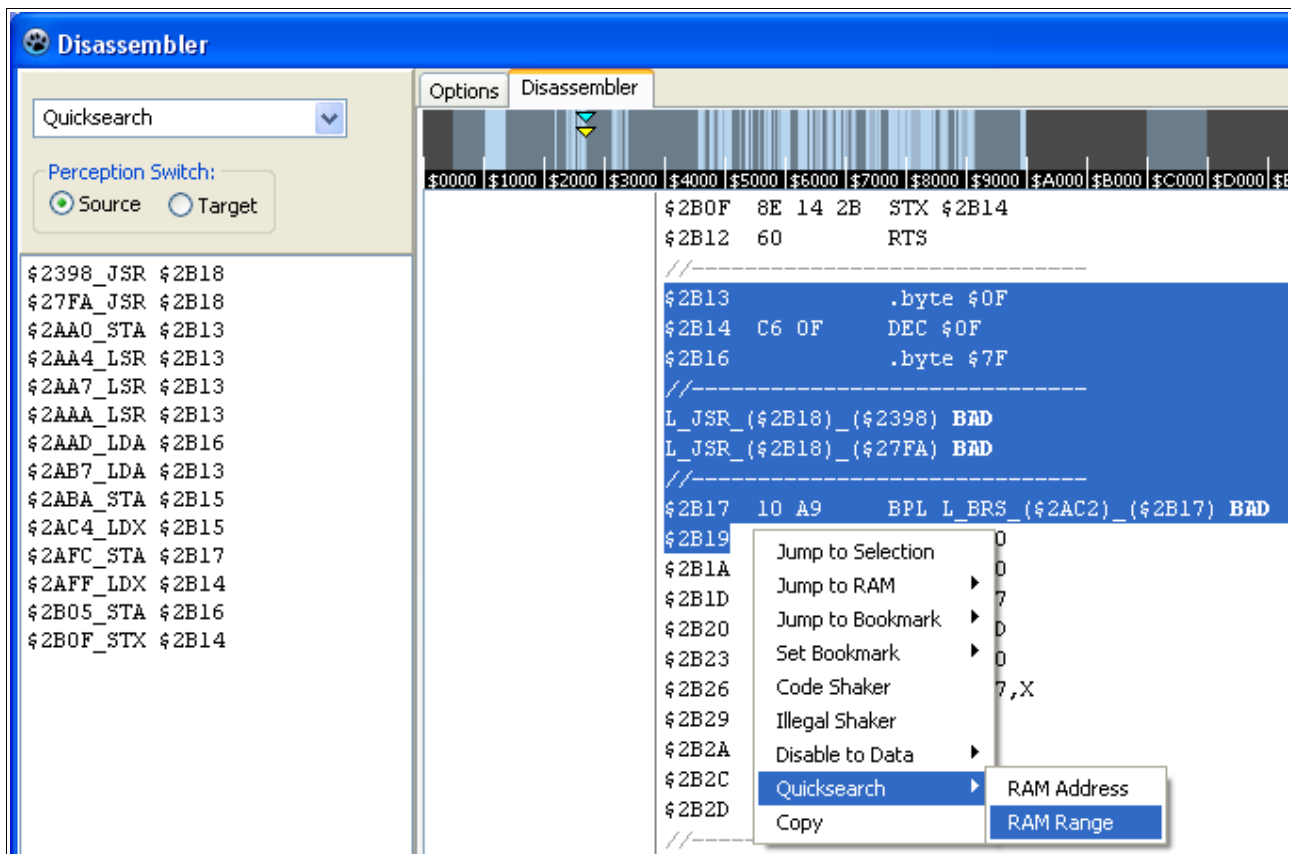
```

5.8 Quicksearch and the Quicksearchlist

Quicksearch performs an interpreter scan on the marked offset address, the output is transferred to the Quicksearchlist. Please note that branches will not be handled and not be found.

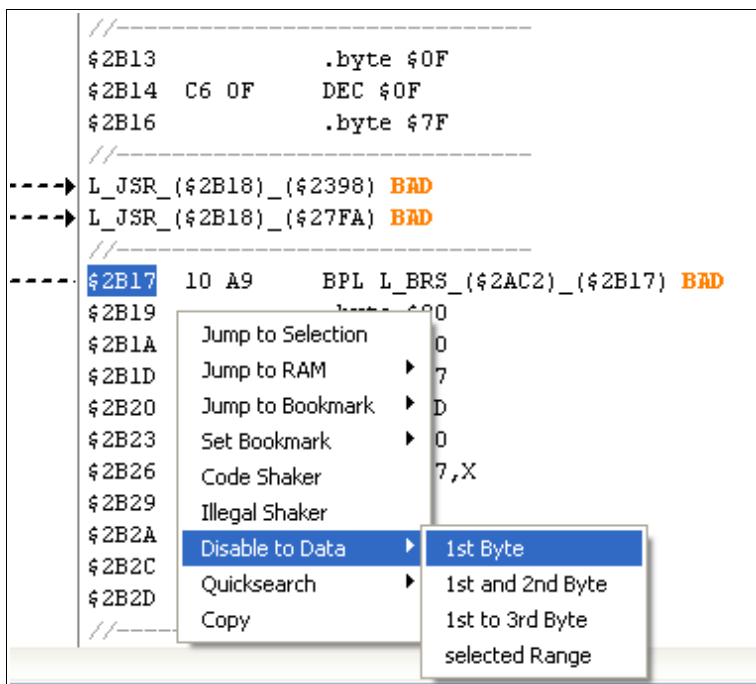


The subitem “RAM Range” is an extended version. This can be very useful when facing a table of pointers. It only accepts a range selection the way shown below. Please note that the last offset address is not part of the range. So in this case, the range is \$2B13 - \$2B18.

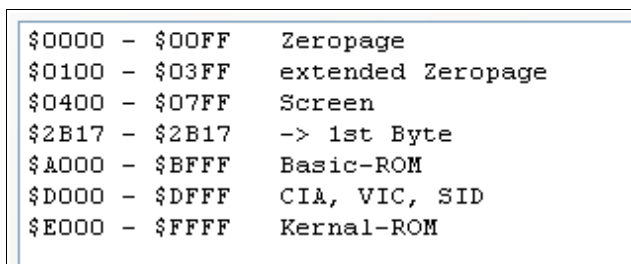


5.9 Disable to Data

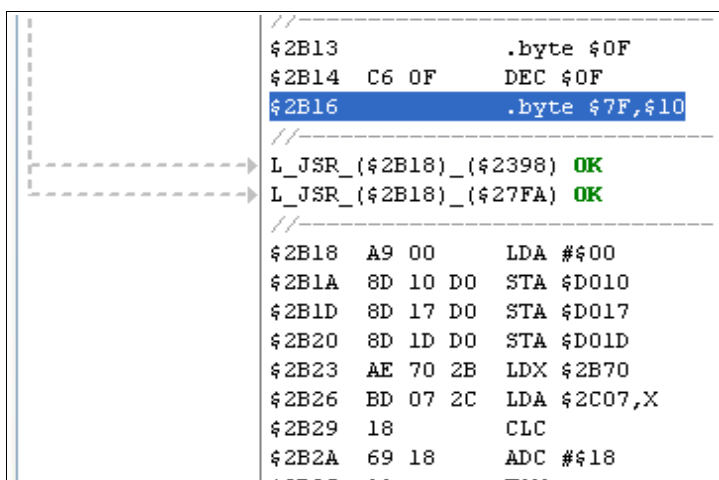
This method can add bytes to the excluded list using the interpreter view. Depending on the option “Autobusy”, a re-disassembling is done immediately. You may use a range instead of handpicking single bytes.



The disabled byte is automatically added to the excluded list.



Result:

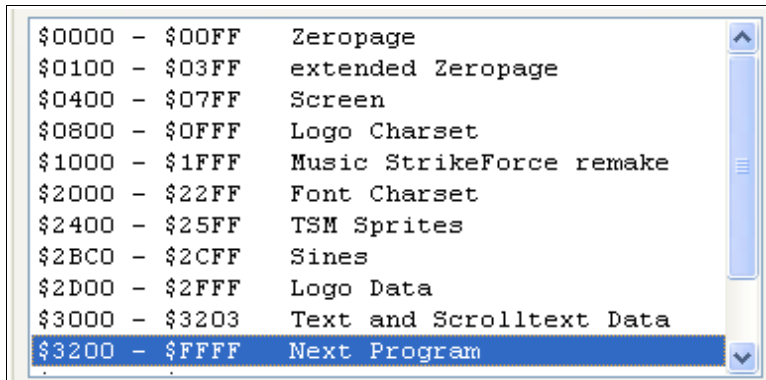


6 Showcase

Welcome to this little showcase. This shall give you some information about how to find several program routines. So let's start with the excluded areas identified in chapter 3.

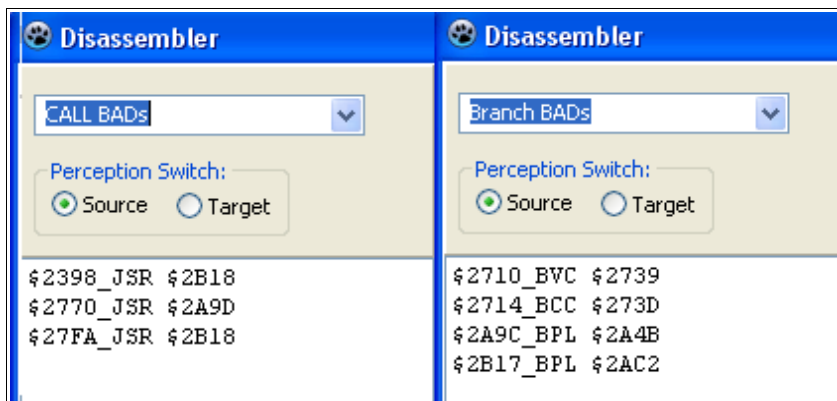
6.1 Fill the Excluded List

In many cases you can never be sure what's really going on – before you really look inside. So, the excluded list is just a draft and not the ultimate final one. You should always put the music on the list. The player, especially it's data, produce additional errors. This is how the excluded list may look like before you push the disassemble button. Because large RAM areas are excluded it shouldn't take longer that one or two seconds.



6.2 Solve the BADs and JAMs

Checking the BAD and JAM entries is the very first thing you should do. Because “JUMPs” and “CALLs” have the tendency to be multiple inherited errors, you should take care of them in the first place. Solving them means to get rid of most problems. Due to the excluded list entry, there are only seven BAD errors and no JAMs. You may get little different results here, it depends on the time your VICE snapshot was made(pointers for movements, color tables and scroll text).



All three “CALLs” are inherited errors triggered by false interpretations at \$2B17 and \$2A9C, so disabling \$2B17 and \$2A9C is a good thing. The \$2B17 problem was discussed in chapter 5, so there's no need to do this again. \$2A9C is pretty much the same, just remember to use the “Perception Switch”, “Codeshaker”, “Quicksearch” and “Disable to Data” for this.

=> Errors at **\$2710** and **\$2714** remaining.

Both errors are close together and in between valid code lines. The indirect “JMP (\$0020)” at \$270B suggests that the errors are not executable code.

```

//-----
L_BRS_($2708)_($2709) OK
//-----
$2708 88 |      DEY
$2709 D0 FD      BNE L_BRS_($2708)_($2709) OK
$270B 6C 20 00    JMP ($0020)
$270E                .byte $1A,$27
$2710 50 27      BVC L_BRS_($2739)_($2710) BAD
$2712 76 27      ROR $27,X
$2714 90 27      BCC L_BRS_($273D)_($2714) BAD
$2716                .byte $B2,$27,$BF,$27
$271A A9 10      LDA #$10
$271C 8D 12 D0    STA $D012
$271F A9 0B      LDA #$0B
$2721 8D 21 D0    STA $D021

```

It looks a lot like an internal table with many \$27 values used for something else . To be sure of that, you can use the “Quicksearch” functionalities. I used the range \$270E to \$2719(\$271A).

Perception switch:

Source Target

\$26FC LDA \$270E,X

\$2701_LDA \$270F,X

	\$0000	\$1000	\$2000	\$3000	\$4000	\$5000	\$6000	\$7000	\$8000	\$9000	\$A000	\$B000	\$C000	\$D000	\$E000
					\$26FB	AA		TAX							
					\$26FC	BD	0E	27	LDA \$270E,X						
					\$26FF	85	20		STA \$20						
					\$2701	BD	0F	27	LDA \$270F,X						
					\$2704	85	21		STA \$21						
					\$2706	A0	05		LDY #\$05						
									//-----						
									L_BRS_(\$2708)_(\$2709) OK						
									//-----						
					\$2708	88			DEY						
					\$2709	D0	FD		BNE L_BRS_(\$2708)_(\$2709) OK						
					\$270B	6C	20	00	JMP (\$0020)						
					\$270E				.byte \$1A,\$27						
					\$2710	50	27		BVC L_BRS_(\$2739)_(\$2710) BAD						
					\$2712	76	27		ROR \$27,X						
					\$2714	90	27		BCC L_BRS_(\$273D)_(\$2714) BAD						

Yes, it's a jump vector table for \$0020/\$0021. => exclude!

A jump vector table is always a nice thing to have. Since Infiltrator can't produce labels for indirect jumps, I got some additional information about the program design. However, they may appear in the “Snippets” list. These are the table values without any code interpretation:

```

$270E      .byte $1A,$27,$50,$27,$76,$27,$90,$27
$2716      .byte $B2,$27,$BF,$27

```

Time to take care of the basic framework.

Page 31 of 39 Pages

6.3 Understand the Program Framework

Catching the start and end of a program is not always that easy. But since we are facing an intro it should not be that hard. I don't like to discuss all attempts, so let's try something simple:

CALL for music player initialization (Interpreter Search 1): CALL at \$27F7, part of a subroutine at \$27EE which is called from \$260B. That routine starts at \$2603 with "LDA \$02A6" (checking for the PAL/NTSC version) and does not have a label. Gotcha!

```

$25F3      .byte $00,$00,$00,$00,$00,$00,$00,$00
$25FB      .byte $00,$00,$00,$00,$00,$00,$02
$2601  0A      ASL A
$2602      .byte $07
$2603  AD A6 02  LDA $02A6
$2606  D0 03      BNE L_BRS_($260B)_($2606)  OK
$2608  2C 00 23  BIT $2300
//-----
L_BRS_($260B)_($2606)  OK
//-----
--- $260B  20 EE 27  JSR L_JSR_($27EE)_($260B)  OK
$260E  A9 04      LDA #$04
$2610  8D E6 26  STA $26E6
$2613  A9 20      LDA #$20
$2615  8D 62 29  STA $2962
$2618  A9 90      LDA #$90

```

Searching for the end: Easy, because "Space" activates the end! Use the "CIA 1" list with register \$DC01, you get the loop for the keyboard scan. You can catch the memory move routine along the way. (LDA \$2698,X → STA \$0400,X → JMP \$040F → LDA \$3204,Y → STA \$0801,Y)

<pre> L_BRS_(\$2670)_(\$2674) OK //----- \$2670 9D 00 D4 STA \$D400,X \$2673 CA DEX \$2674 10 FA BPL L_BRS_(\$2670)_(\$ \$2676 2C 00 80 BIT \$8000 \$2679 A2 00 LDX #\$00 //----- L_BRS_(\$267B)_(\$2687) OK //----- \$267B BD 98 26 LDA \$2698,X \$267E 9D 00 04 STA \$0400,X \$2681 A9 00 LDA #\$00 \$2683 9D 00 D8 STA \$D800,X \$2686 E8 INX \$2687 D0 F2 BNE L_BRS_(\$267B)_(\$ \$2689 A2 00 LDX #\$00 \$268B A9 01 LDA #\$01 //----- L_BRS_(\$268D)_(\$2693) OK //----- \$268D 9D 00 D8 STA \$D800,X \$2690 E8 INX \$2691 E0 0F CPX #\$0F \$2693 D0 F8 BNE L_BRS_(\$268D)_(\$ \$2695 4C 0F 04 JMP \$040F \$2698 09 0E ORA #\$0E </pre>	<pre> \$26A7 A9 34 LDA #\$34 \$26A9 85 01 STA \$01 \$26AB A9 04 LDA #\$04 \$26AD 85 02 STA \$02 \$26AF A9 32 LDA #\$32 \$26B1 85 03 STA \$03 \$26B3 A9 01 LDA #\$01 \$26B5 85 04 STA \$04 \$26B7 A9 08 LDA #\$08 \$26B9 85 05 STA \$05 //----- L_BRS_(\$26BB)_(\$26CC) OK //----- \$26BB A0 00 LDY #\$00 //----- L_BRS_(\$26BD)_(\$26C2) OK //----- \$26BD B1 02 LDA (\$02),Y \$26BF 91 04 STA (\$04),Y \$26C1 C8 INY \$26C2 D0 F9 BNE L_BRS_(\$26BD)_(\$ \$26C4 E6 03 INC \$03 \$26C6 E6 05 INC \$05 \$26C8 A5 03 LDA \$03 \$26CA C9 FF CMP #\$FF \$26CC D0 ED BNE L_BRS_(\$26BB)_(\$ </pre>
---	---

Get IRQ start: Use the “Interrupt” list to get \$26E7

```

$2622 78      SEI
$2623 A9 01    LDA #$01
$2625 8D 1A D0 STA $D01A
$2628 A9 7F    LDA #$7F
$262A 8D 0D DC STA $DC0D
$262D A9 1B    LDA #$1B
$262F 8D 11 D0 STA $D011
$2632 A9 E7    LDA #$E7
$2634 8D 14 03 STA $0314
$2637 A9 26    LDA #$26
$2639 8D 15 03 STA $0315
  
```

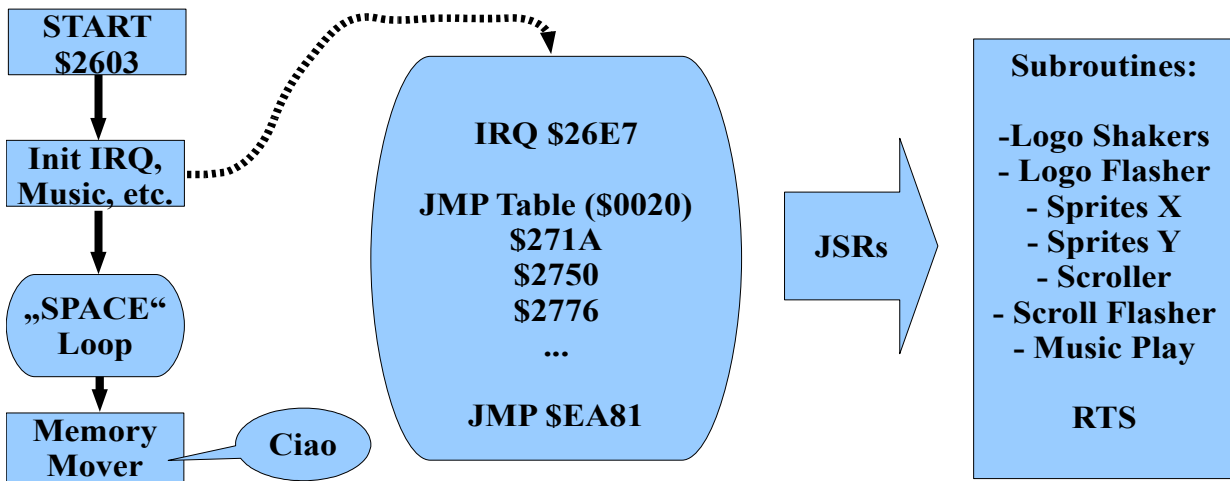
Or use the “Snippets”: Grabs the main start, the IRQ start, the jump vector table addresses seen before and of course some crap.

```

$2347 TXA
$2603 LDA $02A6
$26A7 LDA #$34
$26E7 LDA $D019
$271A LDA #$10
$2750 LDA #$4C
$2776 LDA #$6B
$2790 LDY #$03
$27B2 LDA #$BC
$27BF LDA #$12
$27ED RTI
$28A8 LDX #$0A
$29E1 ORA #$FF
$2A20 ASL $FF
$25E3 .byte $00,$00,$00,$00,$00,$00,
$25EB .byte $00,$00,$00,$00,$00,$00,
$25F3 .byte $00,$00,$00,$00,$00,$00,
$25FB .byte $00,$00,$00,$00,$00,$02
$2601 0A      ASL A
$2602 .byte $07
$2603 AD A6 02 LDA $02A6
$2606 D0 03    BNE L_BRS_($260B)_($2606) OK
$2608 2C 00 23 BIT $2300
//-----
L_BRS_($260B)_($2606) OK
//-----
$260B 20 EE 27 JSR L_JSR_($27EE)_($260B) OK
$260E A9 04    LDA #$04
  
```

You may also try the standard sequence 3 (\$01, \$58) for the final decompression command. It doesn't work here, because the JUMP to \$2603 was originally placed on the screen and is gone. So it depends on the used packer, cruncher and the memory usage. You may receive some decompression code fragments in excluded areas, use “Code Shaker” for a peek.

Intro framework:



6.4 Get the IKARI Logo Shaker

Using the “Screen RAM” list is an efficient approach. The X sine position is read from \$2A9A(calculated somewhere before). Finding the TALENT logo and the scroll is one and the same thing.

The screenshot shows a disassembler window titled "Disassembler" with the "Options" tab selected. The address range is set to "Screen RAM (\$0400)". The left pane lists memory addresses and their corresponding assembly instructions. The right pane shows the disassembled code with a memory map at the top ranging from \$0000 to \$D000. The code includes several instructions, with \$2A51 highlighted in blue. A dashed box highlights a loop structure starting at \$2A42 and ending at \$2A7C.

Address	Hex	Op	Op2	Op3	Op4	Instruction
\$2A3C	AD	98	2A			LDA \$2A98
\$2A3F	8D	9A	2A			STA \$2A9A
\$2A42	20	49	2A			JSR L_JSR_(\$2A49)_(\$2A42) OK
\$2A45	20	7F	2A			JSR L_JSR_(\$2A7F)_(\$2A45) OK
\$2A48	60					RTS
						//-----
						L_JSR_(\$2A49)_(\$2A42) OK
						//-----
\$2A49	AE	9A	2A			LDX \$2A9A
\$2A4C	A0	00				LDY #\$00
						//-----
						L_BRS_(\$2A4E)_(\$2A7C) OK
						//-----
\$2A4E	BD	08	2D			LDA \$2D08,X
\$2A51	99	00	04			STA \$0400,Y
\$2A54	BD	3D	2D			LDA \$2D3D,X
\$2A57	99	28	04			STA \$0428,Y
\$2A5A	BD	72	2D			LDA \$2D72,X
\$2A5D	99	50	04			STA \$0450,Y
\$2A60	BD	A7	2D			LDA \$2DA7,X
\$2A63	99	78	04			STA \$0478,Y
\$2A66	BD	DC	2D			LDA \$2DDC,X
\$2A69	99	A0	04			STA \$04A0,Y
\$2A6C	BD	11	2E			LDA \$2E11,X
\$2A6F	99	C8	04			STA \$04C8,Y
\$2A72	BD	46	2E			LDA \$2E46,X
\$2A75	99	F0	04			STA \$04F0,Y
\$2A78	E8					INX
\$2A79	C8					INY
\$2A7A	C0	28				CPY #\$28
\$2A7C	D0	D0				BNE L_BRS_(\$2A4E)_(\$2A7C) OK
\$2A7E	60					RTS

6.5 Get the Logo Flash Routine

Use the “Graphics” list and try for \$D022 or \$D023, you will find the typical IRQ constructs. Since the color bytes are updated by another routine it's obvious to do a “Quicksearch”.

This routine uses color tables at \$0F10, \$0F50 and \$0F90.

The image shows two screenshots of a disassembler application. The top screenshot shows a list of assembly instructions with a context menu open over the instruction at address \$276D. The bottom screenshot shows the same disassembler with a different view of the code, highlighting a loop structure.

Top Screenshot: Disassembler Interface

- Quicksearch:** [Dropdown]
- Perception Switch:** Source Target
- Assembly List:**
 - \$2362_STA \$276C
 - \$236B_STA \$2767
 - \$2374_STA \$2762
 - \$275E 8D 16 D0 STA \$D016
 - \$2761 A9 06 LDA #\$06
 - \$2763 8D 21 D0 STA \$D021
 - \$2766 A9 0E LDA #\$0E
 - \$2768 8D 23 D0 STA \$D023
 - \$276B A9 03 LDA #\$03
 - \$276D 8D 22 D0 STA \$D022
 - \$2770 20 9D 2A JSR L_JSR_({
 - \$2773 4C 81 EA JMP \$EA81
 - //
- Context Menu:**
 - Jump to Selection
 - Jump to RAM
 - Jump to Bookmark
 - Set Bookmark
 - Code Shaker
 - Disable to Data
 - Quicksearch
 - RAM Address
 - RAM Range
 - Copy

Bottom Screenshot: Disassembler Interface

- Quicksearch:** [Dropdown]
- Perception Switch:** Source Target
- Assembly List:**
 - \$2362_STA \$276C
 - \$236B_STA \$2767
 - \$2374_STA \$2762
 - L_JSR_({234B)_({239F) OK
 - //
 - \$234B CE 92 23 DEC \$2392
 - \$234E AD 92 23 LDA \$2392
 - \$2351 F0 01 BEQ L_BRS_({2354)_({2351) OK
 - \$2353 60 RTS
 - //
 - L_BRS_({2354)_({2351) OK
 - //
 - \$2354 A9 01 LDA #\$01
 - \$2356 8D 92 23 STA \$2392
 - \$2359 A2 18 LDX #\$18
 - \$235B BD 10 0F LDA \$0F10,X
 - \$235E C9 F0 CMP #\$F0
 - \$2360 F0 28 BEQ L_BRS_({238A)_({2360) OK
 - \$2362 8D 6C 27 STA \$276C
 - \$2365 8D E0 27 STA \$27E0
 - \$2368 BD 50 0F LDA \$0F50,X
 - \$236B 8D 67 27 STA \$2767
 - \$236E 8D DB 27 STA \$27DB
 - \$2371 BD 90 0F LDA \$0F90,X
 - \$2374 8D 62 27 STA \$2762
 - \$2377 8D D6 27 STA \$27D6

6.6 Get the TSM Y-Movement Routine

Just use the “Sprites” list and choose. You may wonder about \$2B38 and \$2B3D feeding the Y sprite registers with static values. Well, I don't know – maybe the programmer intended to charge X and Y registers in the same routine.

However, \$2B7A is what we are looking for. The routine is called three times (speeding up the movement) and uses the bouncing sine at \$2BBE.

The screenshot displays a disassembler window with two panes. The left pane, titled "Sprites", shows a list of memory addresses and their contents. The right pane, titled "Disassembler", shows the assembly code for the routine, with a memory map at the top ranging from \$0000 to \$D000. The code includes several jumps and instructions, with the instruction at \$2BAD highlighted in blue.

Left Pane (Sprites):

- \$2867_STA \$D010
- \$286C_STA \$D01C
- \$2871_STA \$D01B
- \$2878_STA \$07F8,X
- \$2885_STA \$D025
- \$288A_STA \$D026
- \$288F_STA \$D027
- \$2892_STA \$D028
- \$2895_STA \$D029
- \$2898_STA \$D02A
- \$289B_STA \$D02B
- \$289E_STA \$D02C
- \$28A1_STA \$D02D
- \$28A4_STA \$D02E
- \$295E_STA \$D015
- \$2B1A_STA \$D010
- \$2B1D_STA \$D017
- \$2B20_STA \$D01D
- \$2B30_STA \$D000,Y
- \$2B33_STA \$D008,Y
- \$2B38_STA \$D001,Y
- \$2B3D_STA \$D009,Y
- \$2B56_LDA \$D010
- \$2B5C_STA \$D010
- \$2BAD_STA \$D001,Y**
- \$2BB3_STA \$D009,Y

Right Pane (Disassembler):

```

$0000 $1000 $2000 $3000 $4000 $5000 $6000 $7000 $8000 $9000 $A000 $B000 $C000 $D000
-----
L_JSR_($2B7A)_($2747) OK
L_JSR_($2B7A)_($274A) OK
L_JSR_($2B7A)_($27FD) OK
//-----
$2B7A AE FF 2B LDX $2BFF
$2B7D A0 00 LDY #$00
$2B7F 20 A7 2B JSR L_JSR_($2BA7)_($2B7F) OK
$2B82 8E FF 2B STX $2BFF
$2B85 AE 00 2C LDX $2C00
$2B88 A0 02 LDY #$02
$2B8A 20 A7 2B JSR L_JSR_($2BA7)_($2B8A) OK
$2B8D 8E 00 2C STX $2C00
$2B90 AE 01 2C LDX $2C01
$2B93 A0 04 LDY #$04
$2B95 20 A7 2B JSR L_JSR_($2BA7)_($2B95) OK
$2B98 8E 01 2C STX $2C01
$2B9B AE 02 2C LDX $2C02
$2B9E A0 06 LDY #$06
$2BA0 20 A7 2B JSR L_JSR_($2BA7)_($2BA0) OK
$2BA3 8E 02 2C STX $2C02
$2BA6 60 RTS
//-----
L_JSR_($2BA7)_($2B7F) OK
L_JSR_($2BA7)_($2B8A) OK
L_JSR_($2BA7)_($2B95) OK
L_JSR_($2BA7)_($2BA0) OK
//-----
$2BA7 BD BE 2B LDA $2BBE,X
$2BAA 18 CLC
$2BAB 69 6F ADC #$6F
$2BAD 99 01 D0 STA $D001,Y
$2BB0 18 CLC
$2BB1 69 15 ADC #$15
$2BB3 99 09 D0 STA $D009,Y
$2BB6 E8 INX
$2BB7 E0 40 CPX #$40
$2BB9 D0 02 BNE L_BRS_($2BBD)_($2BB9) OK
$2BBE A2 00 LDX #$00
//-----
L_BRS_($2BBD)_($2BB9) OK
//-----
$2BBD 60 RTS

```

6.7 Get the Scroll Text Flasher

Use the “Color RAM” list. The routine uses a small color table and has a delay of three frames.

The screenshot shows a disassembler window titled "Disassembler". On the left, there is a list of memory addresses and their contents, with "\$2A0A STA \$DA58,X" highlighted. The main pane shows assembly code with a flowchart. The code includes labels like L_JSR, L_BRS, L_JMP, and L_BRS, with various instructions such as INC, LDA, CMP, BEQ, RTS, LDX, STX, and STA. The preview pane at the bottom shows a snippet of code starting with "BNE L_BRS_(\$2A0A)_(\$2A10) OK".

Address	Code	Comment
\$2683	STA \$D800,X	
\$268D	STA \$D800,X	
\$2833	STA \$DAA8,X	
\$2836	STA \$D918,X	
\$2854	STA \$D9B8,X	
\$2857	STA \$DA58,X	
\$285A	STA \$D968,X	
\$285D	STA \$DA08,X	
\$2998	LDA \$D9B8,X	
\$299B	STA \$D9B9,X	
\$29A3	LDA \$D969,X	
\$29A6	STA \$D968,X	
\$29A9	STA \$DA08,X	
\$29C8	STA \$D9B8	
\$29CB	STA \$DA2F	
\$29CE	STA \$D98F	
\$2A0A	STA \$DA58,X	
\$29E3	EE 13 2A	INC \$2A13
\$29E6	AD 13 2A	LDA \$2A13
\$29E9	C9 03	CMP #03
\$29EB	F0 01	BEQ L_BRS_(\$29EE)_(\$29EB) OK
\$29ED	60	RTS
\$29EE	A9 00	LDA #00
\$29F0	8D 13 2A	STA \$2A13
\$29F3	EE 14 2A	INC \$2A14
\$29F6	AE 14 2A	LDX \$2A14
\$29F9	BD 15 2A	LDA \$2A15,X
\$29FC	C9 FF	CMP #FF
\$29FE	D0 08	BNE L_BRS_(\$2A08)_(\$29FE) OK
\$2A00	A2 00	LDX #00
\$2A02	8E 14 2A	STX \$2A14
\$2A05	4C F6 29	JMP L_JMP_(\$29F6)_(\$2A05) OK
\$2A08	A2 00	LDX #00
\$2A0A	9D 58 DA	STA \$DA58,X
\$2A0D	E8	INX
\$2A0E	E0 28	CPX #28
\$2A10	D0 F8	BNE L_BRS_(\$2A0A)_(\$2A10) OK
\$2A12	60	RTS
\$2A13	01 01	ORA (\$01,X)
\$2A15	06 04	ASL \$04
\$2A17	0E 03 0D	ASL \$0D03
\$2A1A	01 01	ORA (\$01,X)
\$2A1C	0D 03 0E	ORA \$0E03
\$2A1F		.byte \$04
\$2A20	06 FF	ASL \$FF

7 Appendix

7.1 FAQ

- I'm using VICE 2.2! Does it work anyway? The VICE Development Team made some major changes, but I guess it will work. However, I recommend to get the 2.3 version.
- Where are the illegal OP-Codes? I think it does not make sense to do that for the complete memory. Use the illegal shaker in case you think you are facing them.
- What about other emulator imports? One day, maybe.
- Can I have several program instances? Yes!
- Where is the OP-Code "BRK"? The OP-Code \$00 is internally handled as an unknown CPU instruction, so it can be put in rows by the CONCAT. Hope you don't mind too much.
- What about IRQ labels? Simple forward interpretation could result in incomplete or even incorrect results. I don't know how to solve this without writing an OP-Code emulator yet.
- What about generating labels for static offsets (LDA,STA,etc. \$XXXX): Might be useful in case you like to rip off speed code, but also may result in tens of thousands useless labels. However, I guess this function will be used rarely. So... maybe.
- What about an extended CRAP version with IF ELSE and LOOP commands? Maybe.
- Why does Infiltrator use so much RAM? Blame the Lazarus Development Team. At least I used UPX 3.07 to compress the executable file.
- What about function trees? Planned.

7.2 Known Bugs

- The connecting code lines in the disassembler are not redrawn when using the mouse wheel. Until now they are only drawn on a canvas element when pushing the cursors, page up/down keys or the mouse buttons.
- When closing a tool window not using the main Infiltrator form buttons you have to push the button twice for a reinitialisation.
- The CRAP error handling is insufficient.

7.3 "AS IS" Warranty Statement

ATTENTION: BY DOWNLOADING AND USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE FOLLOWING TERMS. IF YOU DO NOT AGREE TO ALL OF THESE TERMS, DO NOT DOWNLOAD AND USE THE SOFTWARE ON YOUR SYSTEM.

"AS IS" WARRANTY STATEMENT

DISCLAIMER. TO THE EXTENT ALLOWED BY LOCAL LAW, THIS SOFTWARE PRODUCT ("SOFTWARE") IS PROVIDED TO YOU "AS IS" WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, WHETHER ORAL OR WRITTEN, EXPRESS OR IMPLIED. THE OWNER OF THE COPYRIGHT SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.

LIMITATION OF LIABILITY. EXCEPT TO THE EXTENT PROHIBITED BY LOCAL LAW, IN NO EVENT WILL THE OWNER OF THE COPYRIGHT BE LIABLE FOR DIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR OTHER DAMAGES (INCLUDING LOST PROFIT, LOST DATA, OR DOWNTIME COSTS), ARISING OUT OF THE USE, INABILITY TO USE, OR THE RESULTS OF USE OF THE SOFTWARE, WHETHER BASED IN WARRANTY, CONTRACT, TORT OR OTHER LEGAL THEORY, AND WHETHER OR NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.