Vic20emu User Manual

Table of Contents

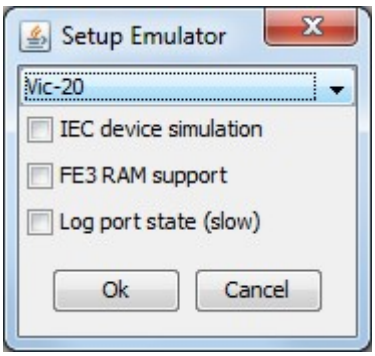
Introduction

The Vic-20 Emulator & Debugger focuses on features that help explore the Vic-20 hardware or at least it helped me understanding the machine better than I did 20 years ago. The internal structure of the Java program resembles the hardware structure, which is not best for performance, but has other benefits with regard to program analysis and modularity.
If you just want to fire up an emulator and play a few old games there are certainly better and more complete alternatives, e.g. Vice[1] or MESS[2]. Also for writing small programs the "Vic20 CBM .prg Studio"[3] may provide you with all you need. Finally, you need a decent PC to run the emulator at reasonable speed.
So why did I feel the urge to write another emulator, if there already are so many very good programs available? Here is why:

- vic20emu is written in Java, so it supports a wide range of host platforms.
- it comes with a modern user-interface that is superior to original debuggers on the Vic-20 and supports analyzing different aspects of the system (e.g., charts for signals on the VIA ports, advanced code profiling).
- the debugger that can be started from the development environment with symbols loaded and ready to run the program, thereby conveniently completing the code-compile-debug cycle.

This document gives a brief overview of the features of the emulator and starts with a quick introduction, followed by descriptions of the modules you may want to use to explore the Vic-20 in greater depth after getting acquainted with the basic functionality of the emulator.

Running the Emulator



When you start the emulator using the setup dialog (which is default), a dialog box lets you choose a few configuration options. Use the drop-down list to select the machine you want to emulate. At present, there is only Vic-20 and an experimental version of the Vic-1541. Vic-1541 emulation does emulate the CPU and the I/O chips, so you can inspect the firmware of the drive for educational purposes, but that is all for now (see Section "Vic 1541 Emulation & Future work").

The options for the Vic-20 emulation are:

- IEC drive simulation (Printers & Floppies)
- FE3 RAM support (512K RAM in Super RAM mode)
- Port state logging (I/O inspection)

Use "IEC device simulation" if you want to load files from your PCs file-system. "FE3 RAM support" is useful to develop or run programs for the Final Expansion 3, like a RAM disk or the Task Switcher. "Port state logging" records the signals of the VIAs I/O (which makes emulation slower) and is explained in Section "I/O inspection".

1 http://www.viceteam.org/
2 http://www.mess.org/
3 http://www.ajordison.co.uk/

Getting Started

When you start the emulator for the first time, the basic windows of the emulator pop up on your desktop and you have to arrange the windows to your needs. Window positions are saved, when you close the application via *Exit* in the file menu (see Figure 1). After you finished moving around and resizing the windows, select *Run* from the CPU menu and the emulator starts running.

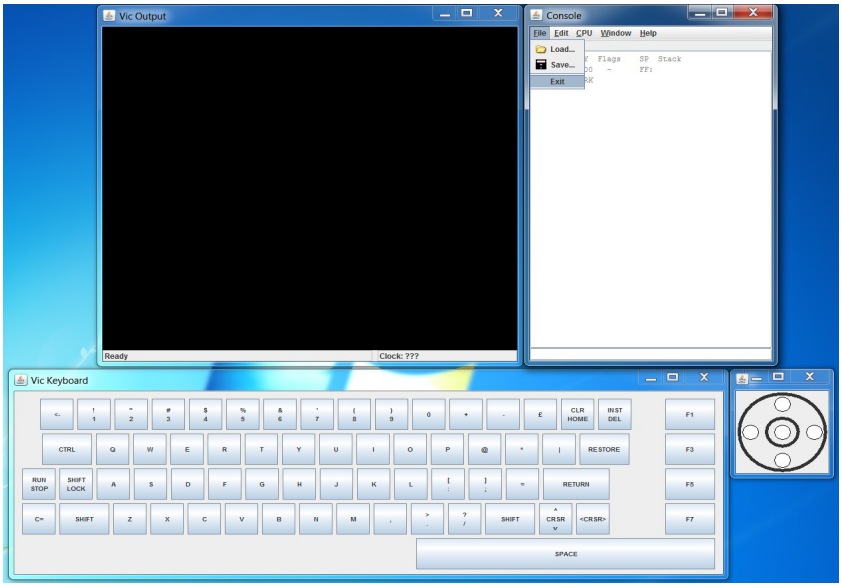
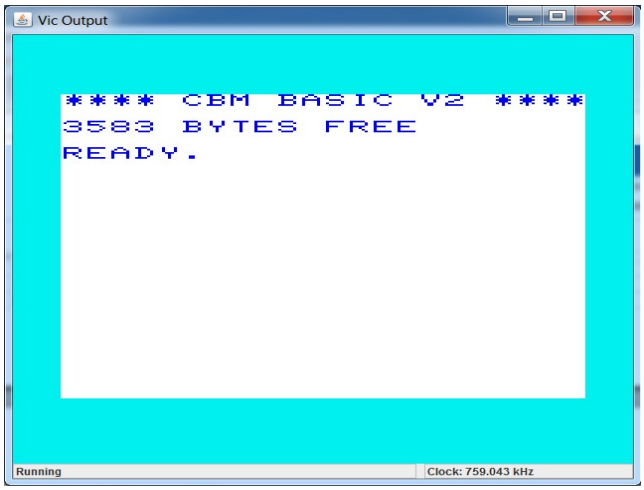


Figure 1: The main windows of the emulator. Close any of the four windows to close the application. Use 'Exit' from the File menu in order to store window positions.

The four main windows of the emulator are the console window and three peripherals windows for screen, keyboard and joystick. Whenever you close one of the four main windows, the emulator quits running.

Vic Output (screen)

The window displays screen output from the VIC once the CPU is running. The status bar at the bottom of the window shows running state and frequency of the emulated CPU.

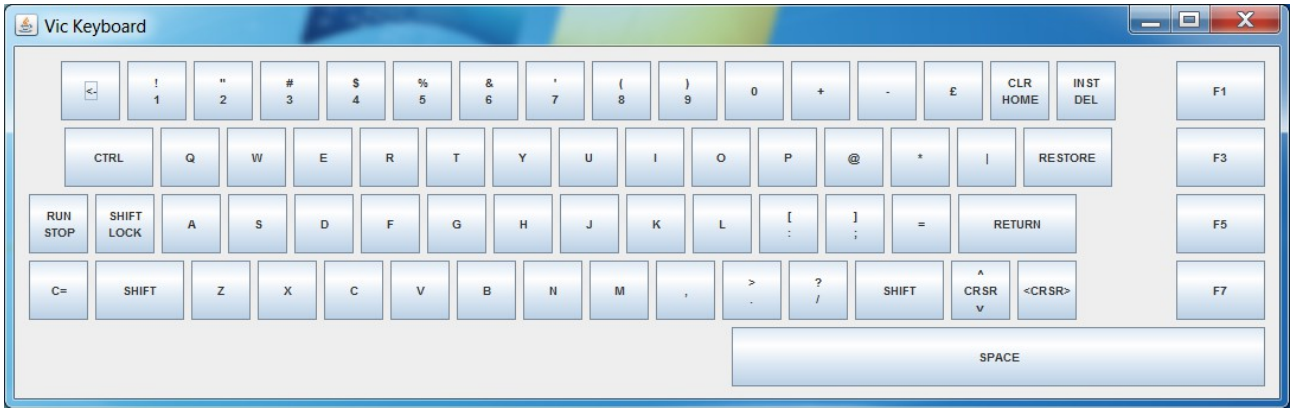


You can use your PC keyboard to type text, when the window is active and you can paste text from other sources into the Vic-20 keyboard buffer, by clicking the right mouse button inside the window.

The numerical keypad and right Ctrl key emulate the Joystick.

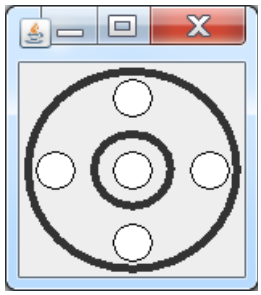
Vic Keyboard

The keyboard window is an imitation of the Vic-20 keyboard. You can use it for keys that are not present on the PC keyboard.



Keys are locked by clicking them with the right mouse button and unlocked by right-clicking them again. This is useful if you want to press multiple keys at the same time or want to freeze keyboard state while hitting a breakpoint or single-stepping through your code.

Vic Joystick

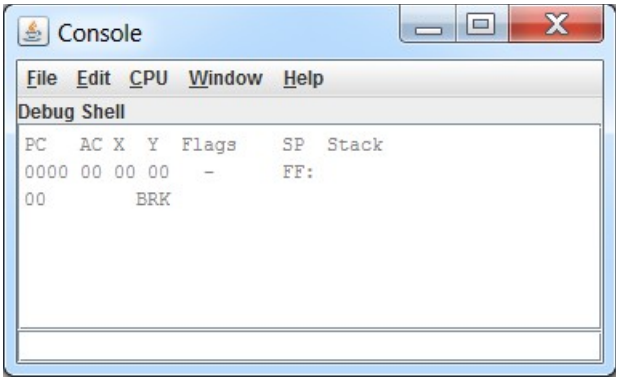


The Joystick window has four small circles for the joystick positions in the outer circle and a circle in the center of the window that shows the state of the fire button.

Move the mouse cursor to the inner circle to "steer" the joystick by the position of the mouse pointer. Click the right mouse button to "fire".

Console

The console window contains menus to access commands, other windows, and the debugger shell that shows CPU messages and a command interface (input field at the bottom). You do not have to type in commands, because most (and more) functions are accessible from the menu or the debugger window. But if you prefer typing instead of clicking you can get a list of the available commands by typing "help". All commands can be used in startup scripts (see chapter "Startup Scripts").

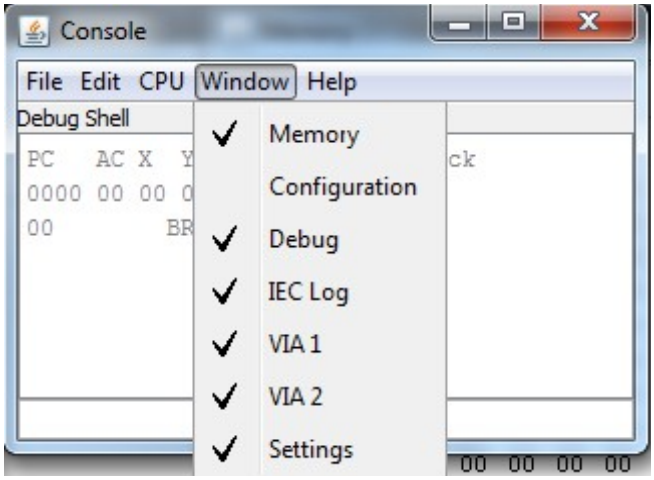


From the **File** menu you can load files into Vic-20 memory or store memory contents to files in different formats.

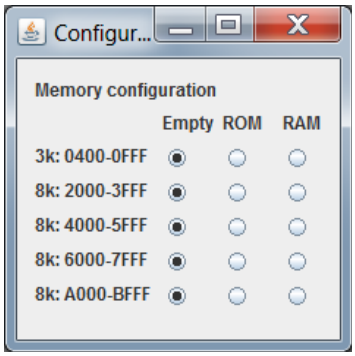
The **Edit** menu is used for command interface input.

The **CPU** menu lets you start, stop and reset the CPU.

The **Window** menu is used to access other useful windows, like the memory *configuration*, a *memory* viewer and editor, and the *debugger* window. The available entries depend on the settings you used when starting the emulator, e.g., "IEC Log" is only available, when the "IEC device simulation" is active.



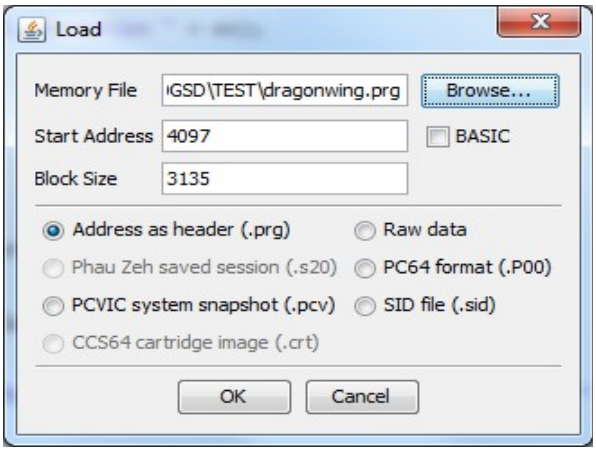
Running a Program



You can load programs directly to memory or use the disk emulation for loading them in the emulator. In both cases you have to select the memory configuration for the program to run. Open the *configuration* window from the *Window* menu and set the memory layout appropriately.

Initially, all external memory areas are switched off. If you set an area to ROM, the Vic-20 cannot write to it, but you can still load files to memory from the File menu. This is useful to prevent ROM programs from accidentally or intentionally overwriting their code.

The *Open* command from the file menu lets you select a file from your filesystem and tries to guess start address, size and format, if possible.



For example, the parameters of the file "dragonwing.prg" are guessed correctly after selecting it with the "Browse..." button. The option "BASIC" is necessary for BASIC programs that are loaded to an address that differs from their original location.

After loading the program to memory you can switch to the Output window, type RUN and hit "Enter" to start for example this great program written by Aleksi Eben. (Unfortunately, the emulator does not play sound yet)

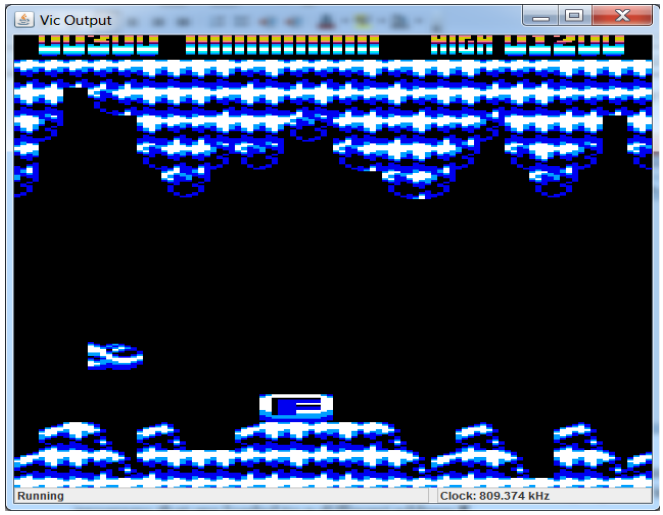
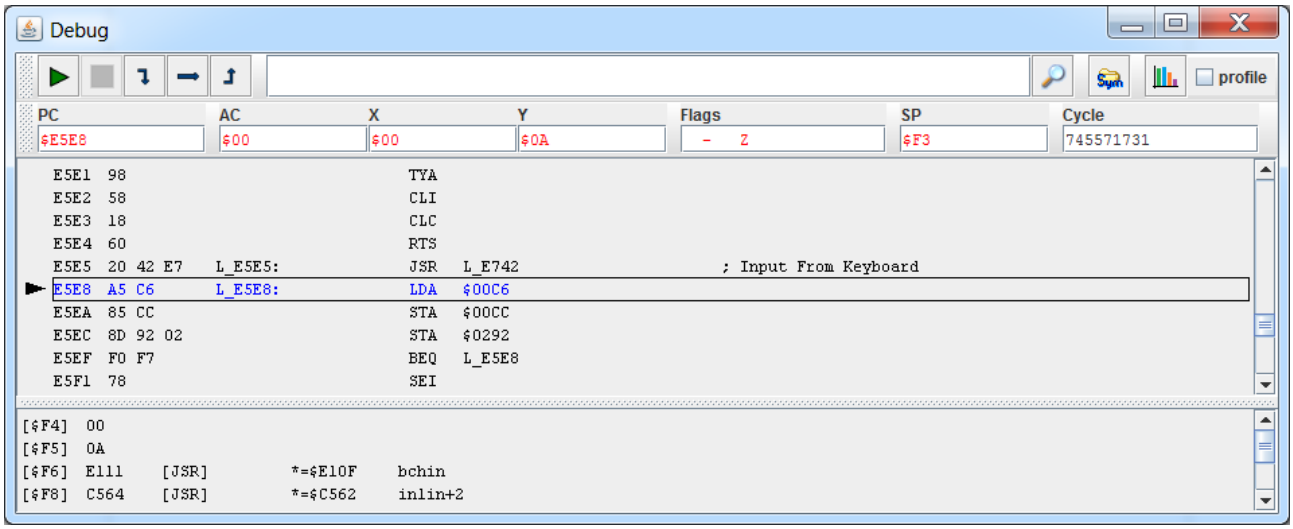


Figure 2: Dragon Wing running in the emulator.

The Debugger Window

You can access the debugger window from the console window (Window menu).

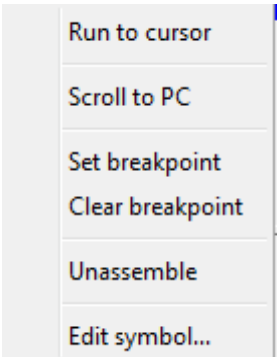


Here you can

- Control execution
- Search for code locations (addresses and labels)
- Load, edit and save symbols
- Enable profiling and view profiling results
- Review and modify register values
- Browse through code
- Set and clear breakpoints
- Navigate to return addresses on the stack.

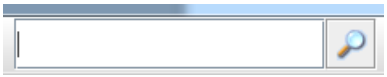
You can start and stop execution as well as single step through the program by using the buttons in the toolbar. The arrow pointing down steps into subroutine calls and interrupts, whereas the straight arrow stops execution after returning from a JSR. The up arrow stops after the next RTS/RTI.

Another way to control execution uses the context menu. Click the right mouse button inside the code area and a menu pops up.



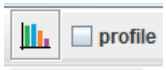
Using this menu you can continue execution until it reaches the selected line, or set and clear breakpoints.

Breakpoints are marked by a red bullet to the left of a code line and can also be set or cleared by a double click in that area. (The black triangle marks the current position of the program counter.)

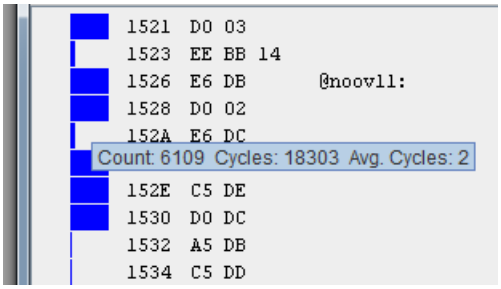


Right next to the command buttons is the search box. You can enter a label or an address and hit enter or the search button to navigate to the corresponding location in memory. (Hex numbers need a '$' in front, as in "$FFD2"). If the debugger can not find the label or address the search box turns orange until you type something else into the search box.

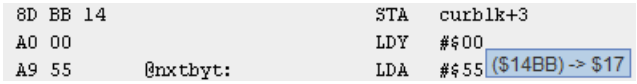
With the "Symbols"-Button, you can open the symbol window, where you can load and save symbols to and from file (e.g. labels from ca65) (see Section "Symbols").



The checkbox labeled "profile" enables execution profiling. When enabled, the debugger displays a blue bar in every row in the code window. The width of the bar correlates with how often the statement was executed. Move the mouse pointer over the bar to get more results as a tool tip or review the results in another window with a click on the small bar graph in the toolbar (see Section "Profiling").

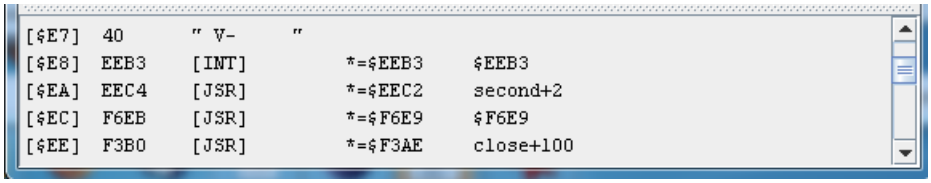


You also get a tool tip when you move the mouse pointer over the operand of a statement. For example, the operand "curblk+3" of the statement STA has the tooltip "($14BB) -> $17.



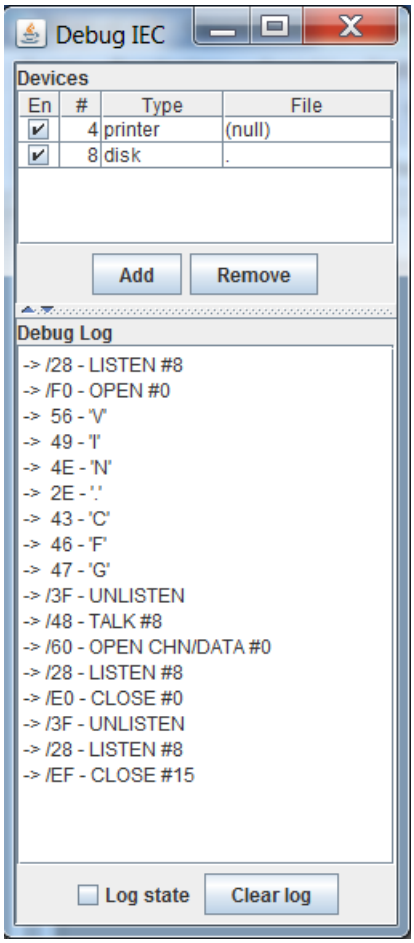
$14BB is the value of curblk+3 and the target address of the store command STA. The current value at address $14BB is $17. This is useful, if you step through code and want to know what is going on in the next step without looking for the correct locations in the memory window.

The second toolbar shows register values, stack pointer, and CPU cycles elapsed since starting the emulator. Red text color indicates that a value has changed since the last break. You can change each of the register values and flags, but not the cycle count.



The stack list in the bottom of the debugger window shows the annotated stack contents. Entries marked with [JSR] or [INT] are pushed by a JSR statement or Interrupt respectively. You can jump to the corresponding location in the code window by double clicking the line in the stack list.

The IEC Window (Debug IEC)



The IEC simulator emulates the communication protocol of Commodore's serial bus. Every byte sent over this bus is logged in the "Debug Log" of the IEC window. Commands bytes are preceded by '/' and the command is shown in plain text.

The upper part of the window lists the configured devices on the bus. You can add and remove devices and edit their properties by clicking into the table.

At present, the emulator supports two devices: printer and disk.

Printer is either like a null device that "consumes" every byte and discards it immediately or – if you specify a file name – writes everything it receives to a file.You can get a listing of your BASIC program by printing it to a file using the BASIC commands "OPEN4,4:CMD4:LIST"

Disk is a file system driver that supports reading files (including "$") from the local file system. You can also change the current directory by using the "CD" command known from SD2IEC.

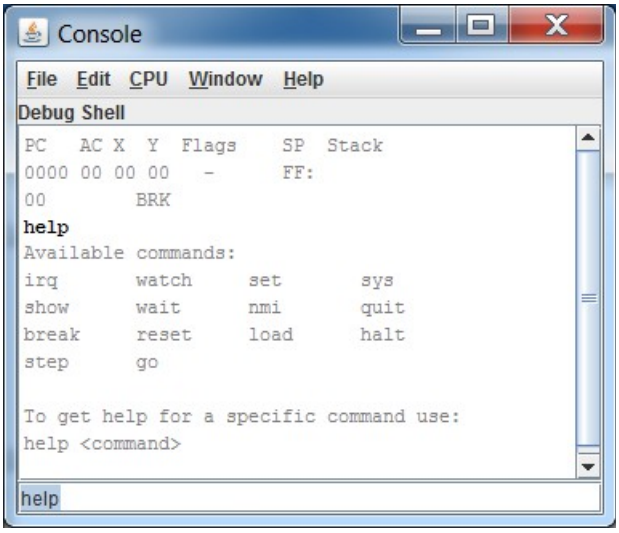
The option "Log State" enables logging of the state transitions of the IEC protocol client simulation and is used to debug the IEC simulation.

Note:

Do not set clock emulation to favor speed over accuracy, if you use the IEC device simulation, because otherwise it will not work correctly.

Startup Scripts

You can use the commands of the debugger console to setup the emulator for the program you want to run or debug. To get a list of available commands type 'help' into the command input line at the bottom of the console window.



You get help for a specific command by typing 'help' followed by the command. To run a specific configuration, write the commands into a file and start the emulator with the file name as parameter.

For example, take a look at the startup script I use to debug VIN[4]:

```
go til ready
wait
set ram 1
set ram 2
set ram 3
load code ../vin/diskmenu.prg
load symbols ../vin/diskmenu.lbl
sys prolog
go til prolog
wait
break set panic2
```

'go til' tells the emulator to run until it hits the given address or label. 'ready' is defined in the ROM symbol table (vicrom.sym) as

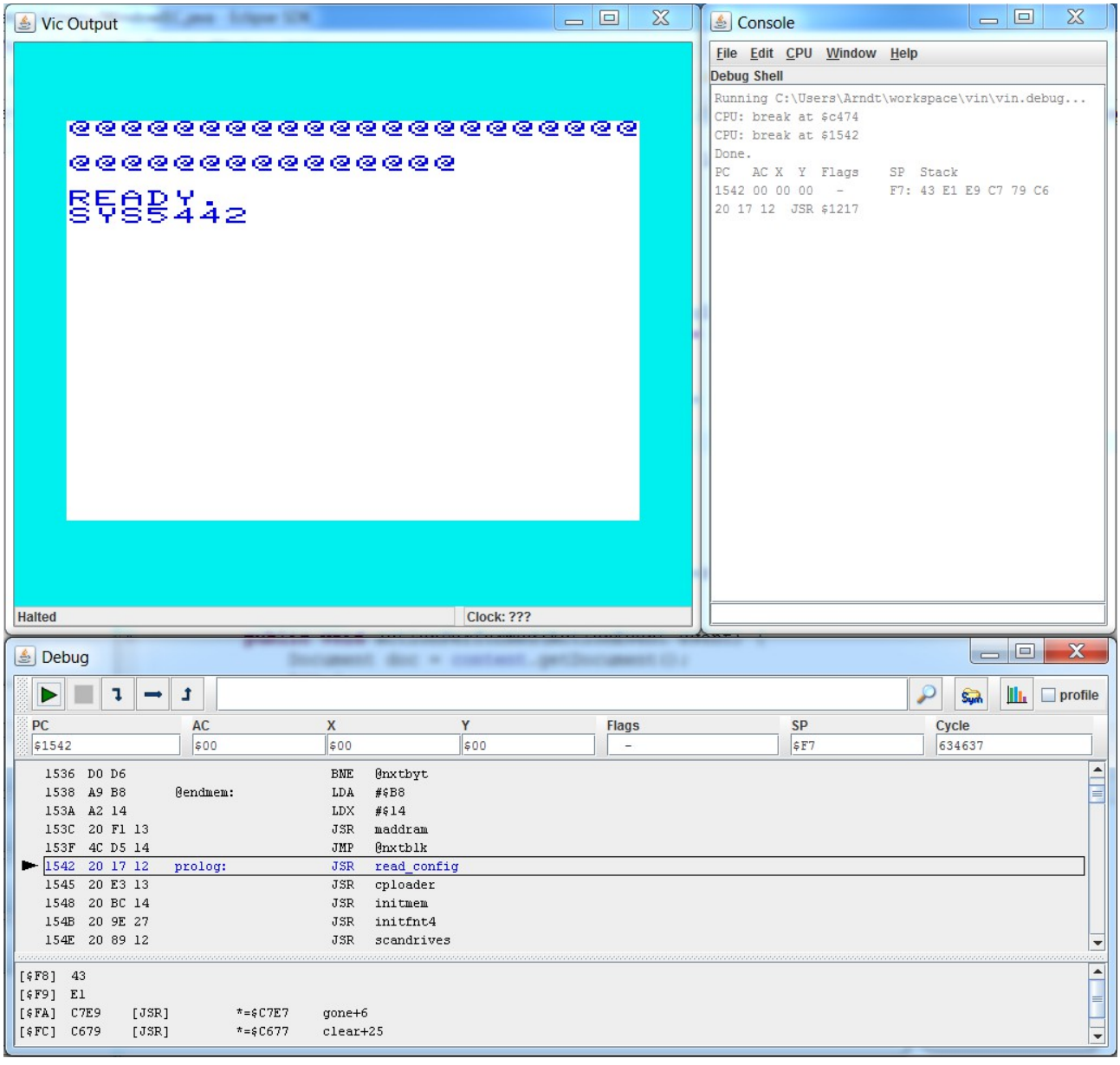
```
c474    ready   Restart BASIC
```

Since the console continues executing commands after 'go', we have to use the 'wait' command to wait until the CPU stops running, before issuing the next commmand. Then, the Vic-20 has finished its initialization and is ready for loading programs. Next, the script enables RAM in banks 1-3 ('set ram') and loads the program ('load code') and its symbols from ca65 ('load symbols').
The programs symbol file defines the symbol 'prolog', which is the entry point of the program. The command 'sys' writes the BASIC 'SYS' command to the keyboard buffer and with 'go til' and 'wait' we let the CPU reach the label 'prolog'.

4 http://code.google.com/p/vin20/

VIN has an error routine that quits the program and prints register values and stack values on a blue screen. While debugging, I prefer to have the program stop at a breakpoint instead. The last line of the script sets this breakpoint at a location in the error routine ('break set').

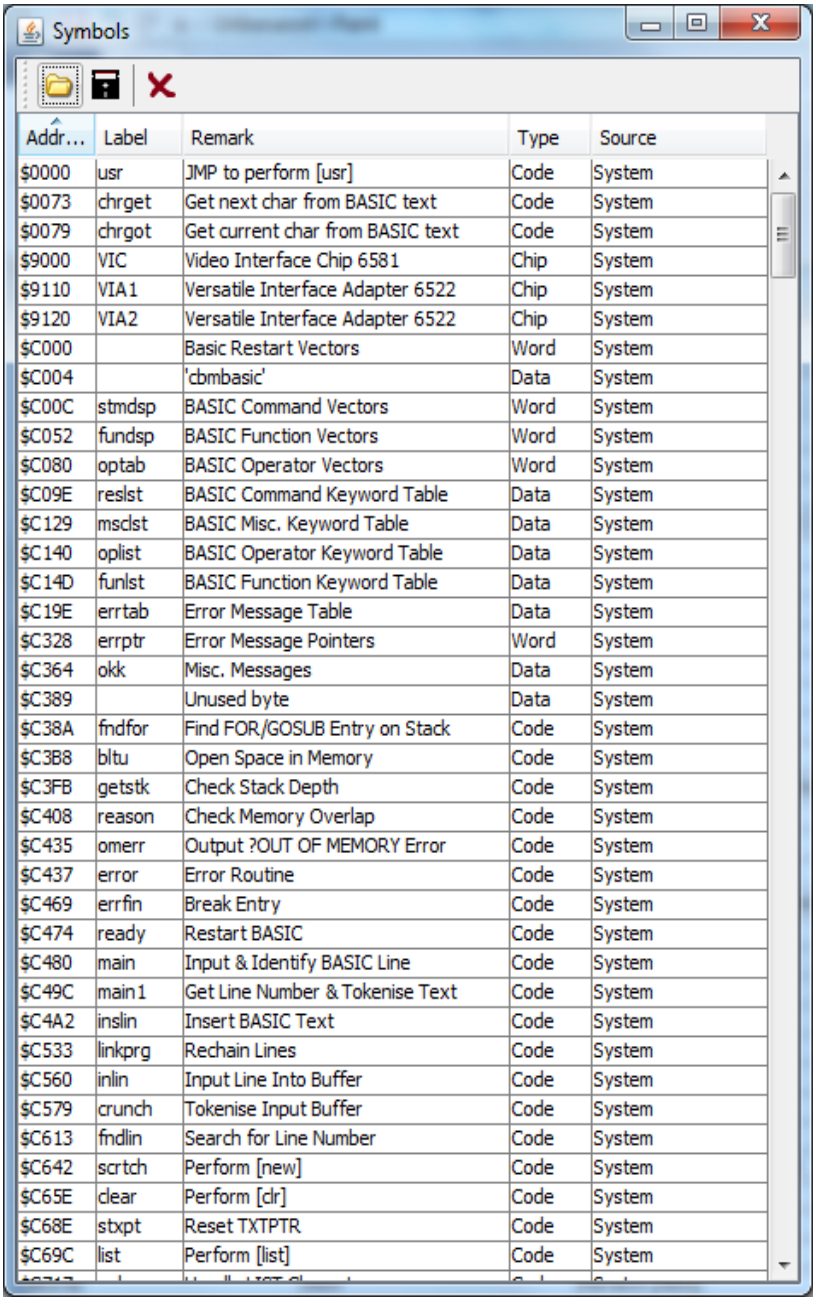
After starting the emulator with the script described above, the PC points exactly to the entry point of my program and the debugger is ready to run.



Symbols

Vic20emu supports using symbols as simple labels generated from ca65 or a combination of label, remark and type specification. The emulator comes with a set of symbols for the Vic-20 ROM derived from the "Commodore VIC-20 ROM Memory Map" found on funet/zimmers and the format of this file is the second type of symbol files the emulator can read.

You can edit symbols in the debugger window and save them to a file for later use using the "Symbols"-button in the debugger's tool-bar. By pressing the "Symbols" button you get a list of all currently known symbols and their source. You can double-click on a symbol to navigate to the respective location in the debugger window.



Addr...	Label	Remark	Type	Source
$0000	usr	JMP to perform [usr]	Code	System
$0073	chrget	Get next char from BASIC text	Code	System
$0079	chrgot	Get current char from BASIC text	Code	System
$9000	VIC	Video Interface Chip 6581	Chip	System
$9110	VIA1	Versatile Interface Adapter 6522	Chip	System
$9120	VIA2	Versatile Interface Adapter 6522	Chip	System
$C000		Basic Restart Vectors	Word	System
$C004		'cbmbasic'	Data	System
$C00C	stmdsp	BASIC Command Vectors	Word	System
$C052	fundsp	BASIC Function Vectors	Word	System
$C080	optab	BASIC Operator Vectors	Word	System
$C09E	reslst	BASIC Command Keyword Table	Data	System
$C129	msclst	BASIC Misc. Keyword Table	Data	System
$C140	oplist	BASIC Operator Keyword Table	Data	System
$C14D	funlst	BASIC Function Keyword Table	Data	System
$C19E	errtab	Error Message Table	Data	System
$C328	errptr	Error Message Pointers	Word	System
$C364	okk	Misc. Messages	Data	System
$C389		Unused byte	Data	System
$C38A	fndfor	Find FOR/GOSUB Entry on Stack	Code	System
$C3B8	bltu	Open Space in Memory	Code	System
$C3FB	getstk	Check Stack Depth	Code	System
$C408	reason	Check Memory Overlap	Code	System
$C435	omerr	Output ?OUT OF MEMORY Error	Code	System
$C437	error	Error Routine	Code	System
$C469	errfin	Break Entry	Code	System
$C474	ready	Restart BASIC	Code	System
$C480	main	Input & Identify BASIC Line	Code	System
$C49C	main1	Get Line Number & Tokenise Text	Code	System
$C4A2	inslin	Insert BASIC Text	Code	System
$C533	linkprg	Rechain Lines	Code	System
$C560	inlin	Input Line Into Buffer	Code	System
$C579	crunch	Tokenise Input Buffer	Code	System
$C613	fndlin	Search for Line Number	Code	System
$C642	scrtch	Perform [new]	Code	System
$C65E	clear	Perform [clr]	Code	System
$C68E	stxpt	Reset TXTPTR	Code	System
$C69C	list	Perform [list]	Code	System

Symbols are organized by their source. The symbols loaded on startup are marked as "System", Symbols you edit in the debugger window as "User" and all other symbols by the file they were loaded from. In order to save a set of symbols, you just have to select a file name and the source group(s) to save.

Profiling

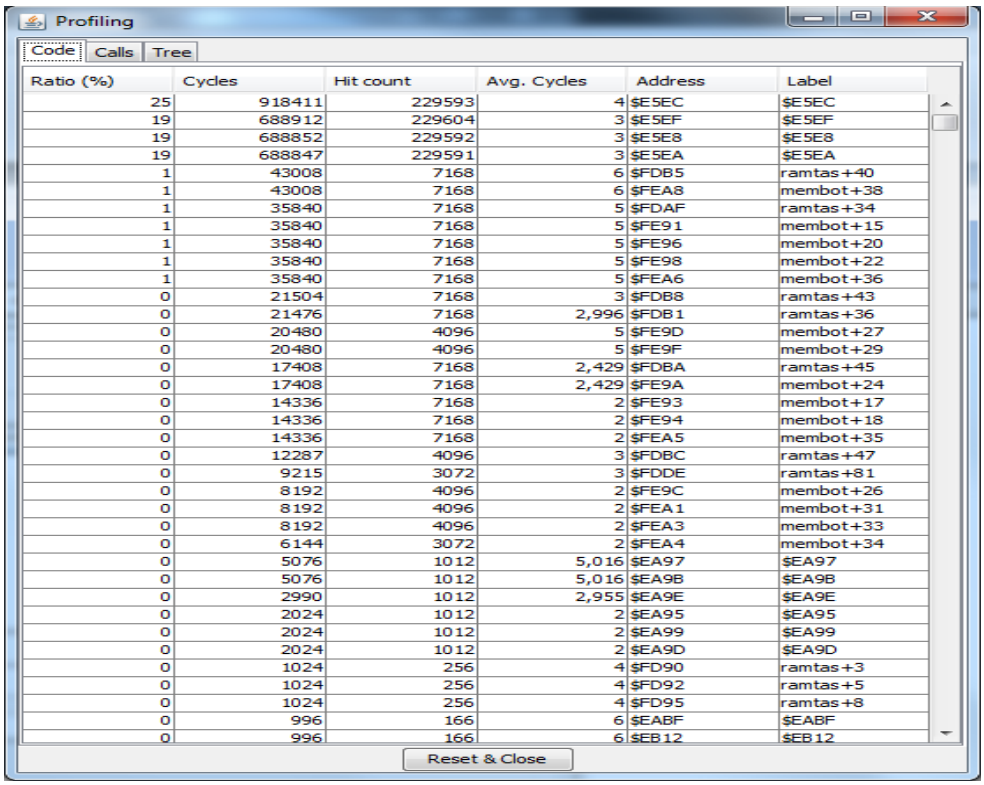
The vic20emu profiler monitors code execution when enabled and generates statistics for executed statements during a profiling run. This results in execution statistics for single statements as mentioned in the description of the debugger window, where you can see immediately how many cycles each of your statements consumed. The debugger window also has a stop-watch for the CPU clock, so you can exactly measure the elapsed cycles between breakpoints.

Since examining the code in the debugger window is certainly not the best way to determine the part of your code that uses most of the execution time, you can open a window with comprehensive results in the "Profiling" window.

The profiling window shows results gathered during profiling runs in three categories:

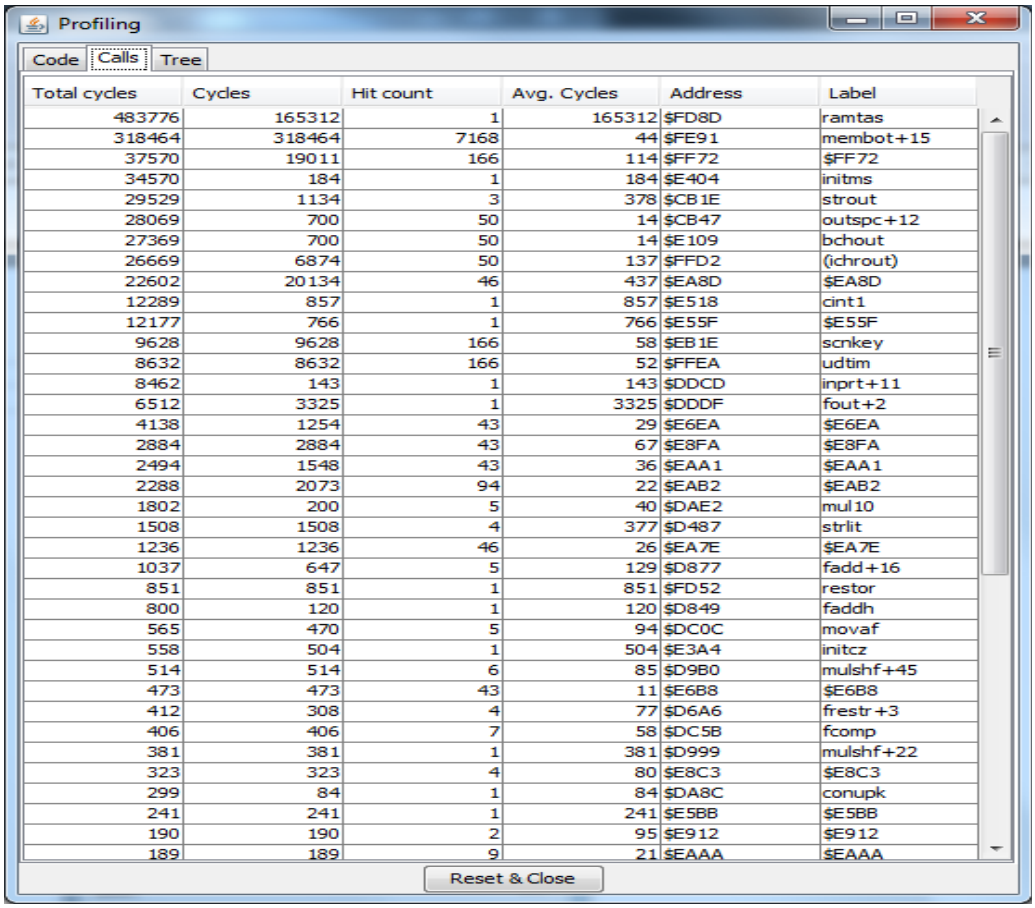
- Code statistics ("Code" tab)
- Function calls ("Calls" tab)
- Call tree ("Tree tab")

Code statistics show the results you already know from the debugger window in a table you can sort by each column with the additional information of the ratio of time spent on the instruction related to the total execution time.



Ratio (%)	Cycles	Hit count	Avg. Cycles	Address	Label
25	918411	229593	4	$E5EC	$E5EC
19	688912	229604	3	$E5EF	$E5EF
19	688852	229592	3	$E5E8	$E5E8
19	688847	229591	3	$E5EA	$E5EA
1	43008	7168	6	$FDB5	ramtas+40
1	43008	7168	6	$FEA8	membot+38
1	35840	7168	5	$FDAF	ramtas+34
1	35840	7168	5	$FE91	membot+15
1	35840	7168	5	$FE96	membot+20
1	35840	7168	5	$FE98	membot+22
1	35840	7168	5	$FEA6	membot+36
0	21504	7168	3	$FDB8	ramtas+43
0	21476	7168	2,996	$FDB1	ramtas+36
0	20480	4096	5	$FE9D	membot+27
0	20480	4096	5	$FE9F	membot+29
0	17408	7168	2,429	$FDBA	ramtas+45
0	17408	7168	2,429	$FE9A	membot+24
0	14336	7168	2	$FE93	membot+17
0	14336	7168	2	$FE94	membot+18
0	14336	7168	2	$FEA5	membot+35
0	12287	4096	3	$FDBC	ramtas+47
0	9215	3072	3	$FDDE	ramtas+81
0	8192	4096	2	$FE9C	membot+26
0	8192	4096	2	$FEA1	membot+31
0	8192	4096	2	$FEA3	membot+33
0	6144	3072	2	$FEA4	membot+34
0	5076	1012	5,016	$EA97	$EA97
0	5076	1012	5,016	$EA9B	$EA9B
0	2990	1012	2,955	$EA9E	$EA9E
0	2024	1012	2	$EA95	$EA95
0	2024	1012	2	$EA99	$EA99
0	2024	1012	2	$EA9D	$EA9D
0	1024	256	4	$FD90	ramtas+3
0	1024	256	4	$FD92	ramtas+5
0	1024	256	4	$FD95	ramtas+8
0	996	166	6	$EABF	$EABF
0	996	166	6	$EB12	$EB12

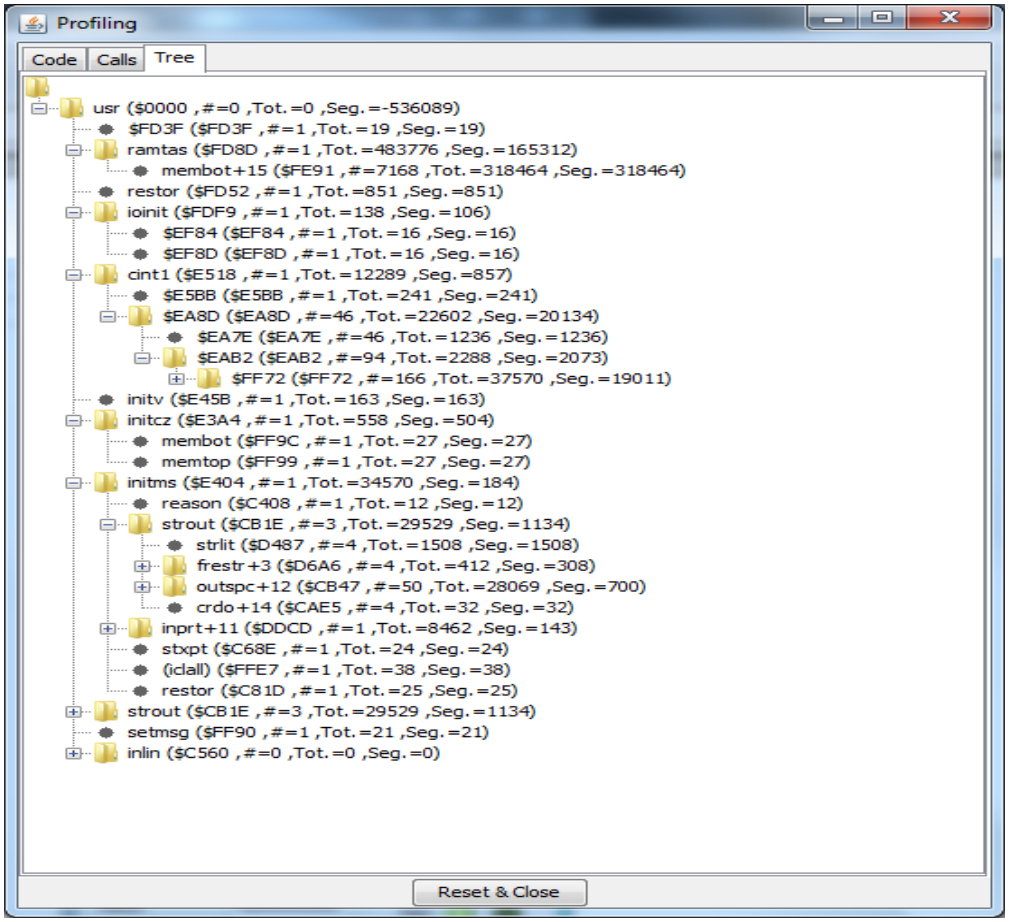
You get a structured view of the profiling results on tab **Function calls**, where execution statistics are counted between function entry ("JSR") and function exit ("RTS").



Total cycles	Cycles	Hit count	Avg. Cycles	Address	Label
483776	165312	1	165312	$FD8D	ramtas
318464	318464	7168	44	$FE91	membot+15
37570	19011	166	114	$FF72	$FF72
34570	184	1	184	$E404	initms
29529	1134	3	378	$CB1E	strout
28069	700	50	14	$CB47	outspc+12
27369	700	50	14	$E109	bchout
26669	6874	50	137	$FFD2	(ichrout)
22602	20134	46	437	$EA8D	$EA8D
12289	857	1	857	$E518	cint1
12177	766	1	766	$E55F	$E55F
9628	9628	166	58	$EB1E	scnkey
8632	8632	166	52	$FFEA	udtim
8462	143	1	143	$DDCD	inprt+11
6512	3325	1	3325	$DDDF	fout+2
4138	1254	43	29	$E6EA	$E6EA
2884	2884	43	67	$E8FA	$E8FA
2494	1548	43	36	$EAA1	$EAA1
2288	2073	94	22	$EAB2	$EAB2
1802	200	5	40	$DAE2	mul10
1508	1508	4	377	$D487	strlit
1236	1236	46	26	$EA7E	$EA7E
1037	647	5	129	$D877	fadd+16
851	851	1	851	$FD52	restor
800	120	1	120	$D849	faddh
565	470	5	94	$DC0C	movaf
558	504	1	504	$E3A4	initcz
514	514	6	85	$D9B0	mulshf+45
473	473	43	11	$E6B8	$E6B8
412	308	4	77	$D6A6	frestr+3
406	406	7	58	$DC5B	fcomp
381	381	1	381	$D999	mulshf+22
323	323	4	80	$E8C3	$E8C3
299	84	1	84	$DA8C	conupk
241	241	1	241	$E5BB	$E5BB
190	190	2	95	$E912	$E912
189	189	9	21	$EAAA	$EAAA

The example above shows typical results for the initialization routine of the Vic-20 kernal ROM. Most of the time was spent in routine "ramtas", which checks and tests RAM available for BASIC programs. It was only executed once, so its "Hit count" is 1 and "Avg. Cycles" equals "Cycles".
If you look at the "cycles" column, you will see that more time was spent in "membot" than in "ramtas". What you do not see in this table is that "membot" is called by "ramtas", but you could guess it from the cumulative call result in column "Total cycles".
"Cycles" represents the CPU cycles spent in a sub-routine without the time spent in calls to other sub-routines. "Total cycles" includes the CPU cycles spent in calls to other sub-routines.
So, since "ramtas" calls "membot+15" and no other sub-routine, the results sum up nicely (318464 + 165312 = 483776). From the third tab ("Tree") you will see, that "ramtas" indeed calls "membot+15".

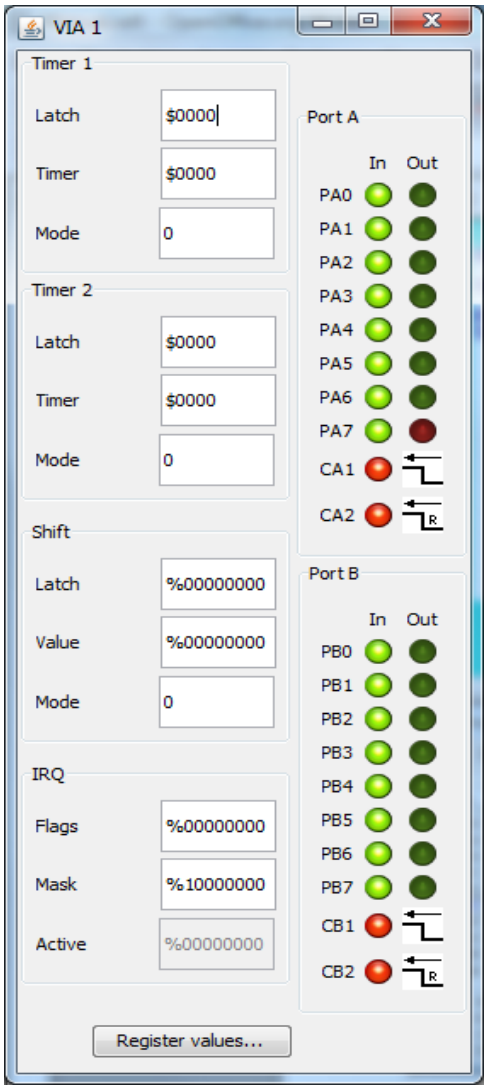
The **Call tree** arranges profiling results by the observed call sequence. Calls are jumps to sub-routines (JSR) or interrupts and the target address of a call is repesented by a sub-node of the caller in the tree.



The profiler generates the call tree only from code that is executed while profiling is active. Results are best, if you start profiling from the outermost code, where all calls are made from. In the example above, profiling was enabled before the CPU started execution for the first time. Hence, the top-level node refers to address 0, which is the initial value of the program counter in the emulator.
The call tree may be inaccurate, if the return address of a sub-routine is removed from the stack without returning to the caller, or if the observed program puts a return address onto the stack by other means than through JSR.
Nevertheless, the call tree is a useful tool, to analyze the structure of a program, especially if you want to get a quick overview of code you do not know yet. You can always double-click on a node of the tree to navigate to its address in the debugger window.

Note: Statistics are only updated, when the profiling window initialises. If you continue profiling while the profiling window is open, it will not refresh. In order to get actual results, close the window and open it again from the debugger window.

I/O inspection



The two VIAs handle peripheral input and output of the VIC-20, i.e., the keyboard, a joystick and the IEC serial bus, which connects the VIC-20 to printers and disc drives. Vic20emu lets you examine the state of the VIAs with respect to I/O, timers and interrupts.

The dialog updates its contents only when the CPU is stopped, so you can not see the timers counting down while the program is running. So this dialog presents a snapshot of the VIAs state.

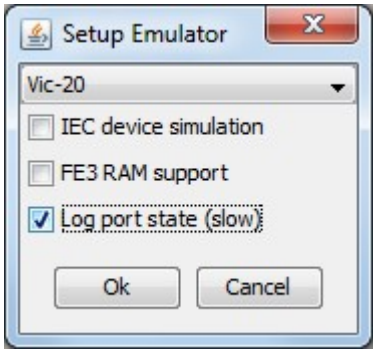
You can not modify the values in this dialog directly, but you can write to the VIA's registers by using the button "Register values...".

The Via 6522 has two 8-bit ports (A and B) for digital input and output. Each of the 8 port lines can be used for input or ouput. If a port line is in **input mode**, the state LED in the "In" row is active. If a port line is in **output mode** its LED in the "Out" row is active (bright red or green).

A **green** LED indicates that the port bit is in logical high state ('1'). The **red** LED corresponds to logical low ("0").

In addition the the 8 port lines each port has two handshake lines (Cx1 and Cx2). Their operating mode is controlled by the port control register (PCR) and the current setting is shown in the small icon right to the LED. Move your mouse cursor over the icon to get a tool-tip with a description of the current mode.

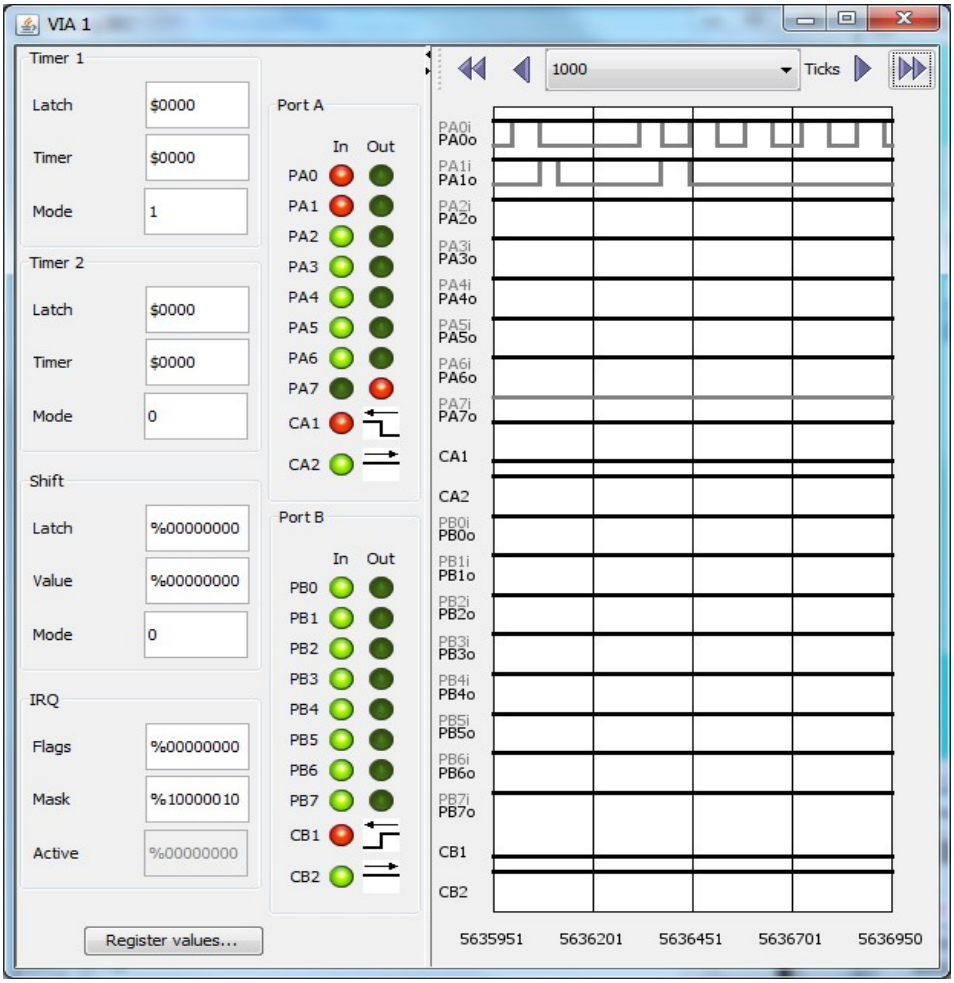
The dialog helps you to debug programs that write to or read from the ports, by showing you the states of the ports at a certain point in you program. For debugging protocols like the serial protocol used for the disk drives and printers, this is only useful, if you almost know, where your program fails, so that you can single step through the suspicious code, while observing the states of the port lines. Sometimes the bug only shows up under certain circumstances, perhaps after transmitting hundreds of bytes and when you notice it, you would have to rewind executing for a lot of cycles to locate its cause.



Vic20emu can do this for you by logging the states of the port lines over a defined number of signal changes. Port state logging slows down execution of the emulation significantly, so you have to enable it in the setup dialog that shows up, when you start the emulator without command line options.

If you enable port state logging, signal graphs are attached to the VIA dialogs and the IEC Simulation dialog ("Debug IEC").

Each of the I/O lines PA0 to PA7 and PB0 to PB7 have to signal graphs. One for its output value and one for its input. For example the input graph for PA0 is labelled PA0i and shown in grey, whereas the output signal is labelled PA0o and drawn in black.



Of course, each port bit is alway either in input or output mode, but its mode can change during execution and by showing the signals with different graphs you can see at a glance which mode the port was in.

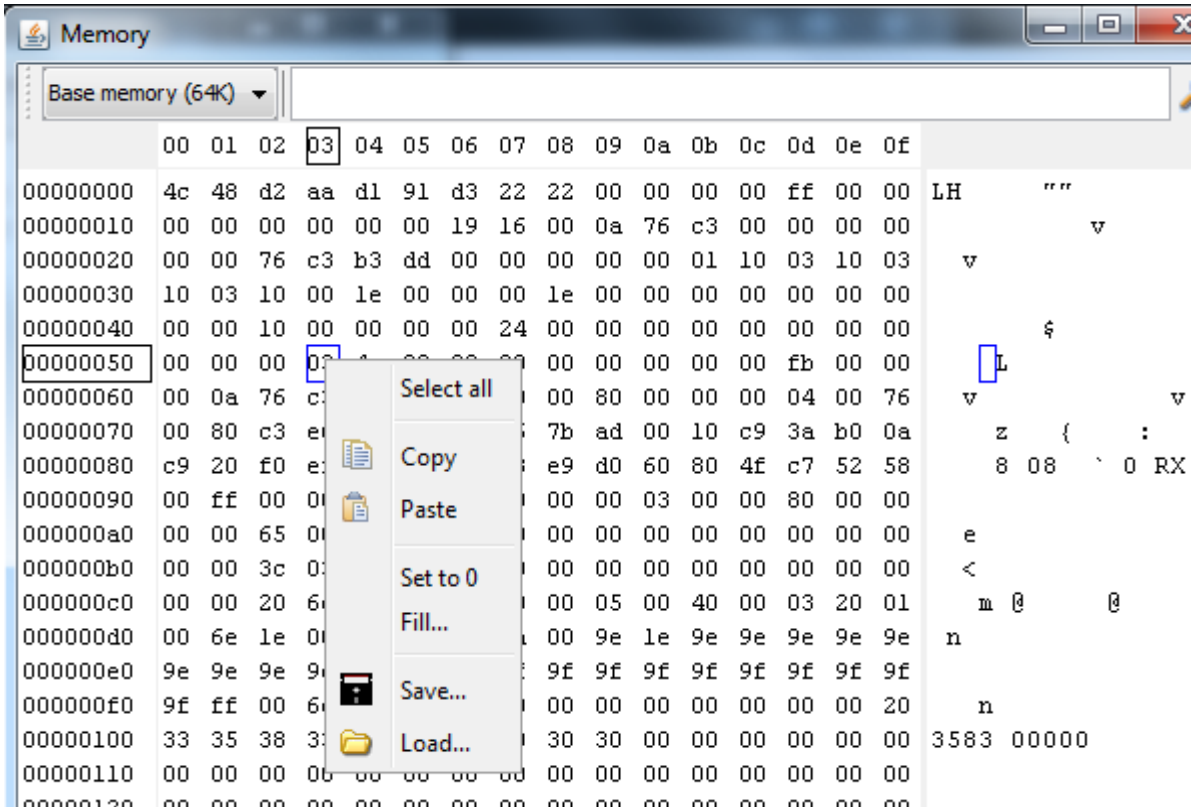
Use the combobox heading the dialog page to select the signal range to display and scroll back and forth in time with the arrow indicators on both sides of the combobox. The time scale is marked in clock ticks elapsed since the start of the emulator, the same value you find in the register panel of the debugger.

Viewing and editing memory contents

Vic20emu has a powerful hex editor for viewing, and manipulating memory data. It shows all RAM that is theoretically available to the CPU, but does not show register values. In FE3 mode the additional 512K RAM is segmented into 16 banks of 32K RAM each, because that is how the memory is organised in Super RAM mode.

Besides viewing and editing memory content directly as byte values or text, you can select memory areas with the mouse (move the mouse while holding the left button) and use the context menu

(right mouse click) to copy, paste, fill, save, or load selected memory regions.



The search box in the top of the dialog lets you search for text or numbers. Move your mouse over the input field to get a description of how to format your seach string.

Vic 1541 Emulation & Future work

For now, the Vic-1541 emulator consists of a CPU, memory and the two VIAs. It lets you debug the drive's code up the the point, where feedback from the I/O ports is needed. For a fully usable implementation, the emulator should also implement the drives hardware to a certain extend. I intend to extend the emulation with respect to the drives hardware and then hook the Vic-1541 emulation up to the Vic-20 emulation thereby replacing the IEC Simulation.

That would form a system that is perfectly suited to examine software that is loaded into the Vic-1541's RAM, like fast loaders, because you could debug both systems at the same time.

Before doing that, I need to improve the execution speed of the emulator, which is not overly difficult, but takes some time. At present speed, emulating two systems at the same time is just not feasible, particularly because I/O needs high accuracy and therefore minimum speed in the present configuration.