wAx 2
6502 Assembler

For Commodore VIC-20

Jason Justian
Beige Maze VIC Lab

January 2022

beigemaze.com/wax2

About wAx

wAx is a machine language monitor for the Commodore VIC-20. wAx runs as a BASIC extension. This means that wAx's commands, including assembly, can be executed from BASIC's direct mode, or seamlessly within BASIC programs.

wAx has all of the usual monitor tools like assembly, disassembly, data entry (hex, text, and binary), breakpoint management, search, transfer and fill, compare, memory load and save, that kind of thing.

But it also has features that are unique or uncommon in native assemblers. wAx is a “somewhat symbolic assembler,” supporting simple symbols and forward references, arithmetic operators, multi-pass assembly capability, and code relocation. It offers built-in support of 6502 “illegal” opcodes. It provides for code unit testing with an assertion tester and selection of multiple BASIC programs in memory. wAx integrates with BASIC by allowing substitution of BASIC variables into wAx commands, including assembly.

wAx also has a user plug-in manager, with a documented API so that you can write your own plug-ins. Or, just use some of the built-in plug-ins included with the package!

wAx is an open-source product, released under the M.I.T. license. It was crafted to be the finest native assembler for the VIC-20 that no-money can buy. I hope you enjoy it.

Live Long and Prosper,

Jason

Acknowledgements

I would like to thank everybody who has supported the development of wAx, especially Michael Kircher, who has helped in too may ways to enumerate. But especially, I owe him debts of gratitude for teaching me about I/O operations, pushing me to perfect stack handling, and contributing to the plug-in environment with his relocation code and “knapsack” debugging ideas, and many other things.

Thanks also to Max Certelli for inventing the *JollyCart* PCB that powers wAxpander. Max helped me finally realize my goals of expanding wAx as a development and debugging platform, while allowing me to bring it to you at a reasonable price. Grazie amico!

About wAxpander

wAxpander is a cartridge that includes one 8K ROM chip containing wAx2 at $A000 (Block 5), and 27K of RAM:

- 3K at $0400-$07FF
- 24K at $2000-$7FFF (Blocks 1, 2, and 3)

When you turn on your VIC-20 with wAxpander installed, the BASIC banner will indicate 28159 bytes free, which includes the on-board 3.5K plus the 24K starting at Block 1. The 3K at $0400 will be available for machine code, or wAx BASIC stages. Screen and color memory will be positioned as usual with 8K+ expansion.

wAx contains software to configure start-up memory. This software, called MEM CONFIG is the default user plug-in when wAx is started from a wAxpander cartridge. To configure memory with MEM CONFIG, do the following:

```
SYS 40960

.U 0K  ; TO RESTART THE VIC-20 AS UNEXPANDED
.U 3K  ; TO RESTART THE VIC-20 WITH 3K
EXPANSION
.U MAX ; TO RESTART THE VIC-20 WITH MAXIMUM
EXPANSION (24K)
```

Notes

- MEM CONFIG restarts the VIC-20 *and* wAx
- Screen and color locations will be appropriate for the configuration selected (e.g., default for 0K and 3K, and moved for 24K)
- Regardless of the BASIC expansion configured, the remaining RAM will be available to you

wAx2 Tutorial

With most native VIC-20 machine language monitors, you're either running the monitor or you're in BASIC. wAx isn't like that. wAx is a system of tools bound together by a common interface that runs as a BASIC extension. It integrates seamlessly into the VIC-20's native environment, providing a great deal of agility. It's a machine language monitor that works *with* you.

If you've used wAx before, there are some important changes in wAx2. If you've used other native VIC-20 assemblers (like VICmon or HES MON), much will be familiar, but wAx does a lot of extra things. So whether you're a new wAx user, or are upgrading to wAx2, the following tutorial will get you up to speed.

Invoking wAx Tools

All wAx tools start with a period (.). The tool's command comes after the period (actually, after any number of periods). In most cases, parameters follow the command, depending on what the tool does. Once you've started wAx, try the following:

```
.?
```

This is the Help Tool. It displays a list of all wAx tools. Each of these can be invoked by entering a period, the command, and usually one or more parameters. Let's invoke a simple tool:

```
.R
```

This shows the register display, and the cursor is now flashing after a period, the wAx prompt. Most tools will finish by showing some kind of prompt. This prompt is there for convenience. It doesn't mean that you're "in" wAx. You never left the VIC-20's BASIC environment.

Disassembly

Now enter this:

```
.D EABF
```

You might recognize this bit of code as part of the VIC-20's interrupt handler. You'll notice that, in addition to the wAx prompt, the cursor is flashing over the D command. Invoking D with only a starting address will display 16 lines of code and then pause. Press RETURN. You'll see *the next* 16 lines of code. Press RETURN again, and hold down the SHIFT key once the code starts rolling. The disassembly will continue until you release SHIFT. You may also disassemble a range of code:

```
.D 0073 008A
```

How do you get out of wAx? Well, you were never "in" wAx to begin with, remember. But you can dismiss the wAx prompt in several ways:

- Press X and then RETURN (the Exit Tool)
- Press DEL until the wAx prompt is gone
- Press the CRSR Down key to go the next line
- Press STOP/RESTORE
- Press SHIFT/CLR

Assembly

This is what you're here for, right? Try entering this:

```
.A 1800 LDA #$2A
```

Once you press RETURN, wAx displays a new kind of prompt. It automatically generates the next A command and populates the next memory address, 1802. Continue entering code:

```
.A 1802 LDY #$10
.A 1804 JSR $FFD2
.A 1807 DEY
.A 1808 BNE $1804
.A 180A RTS
.A 180B {Press RETURN}
```

At the .A 180B prompt, just press RETURN. You'll go back to the normal wAx prompt. Now execute this program with the Go Tool:

```
.G 1800
****************
```

After returning from the program, wAx shows the register display, and a new wAx prompt. Now disassemble your code:

```
.D 1800 180A
```

Cursor up to address $1800, and change the line to LDA #$5C and press RETURN. You may edit your code directly from the disassembly, and comma is an alias for A, the Assemble Tool.

BASIC Assembly

All wAx commands can be used in BASIC programs. Dismiss the wAx prompt (with X or DEL), and enter the following BASIC program:

```
NEW
10 .A 1900 INC $900F
20 .A 1903 JMP $1900
RUN

READY.
SYS 6400
```

The program doesn't do much. It just changes the screen color until you press STOP/RESTORE. This is a simple way to add assembly to BASIC. However, it might be too simple for all but the most trivial uses. Here's a somewhat more advanced example:

```
NEW
10 S = 6400
20 .A 'S   INC $900F
30 .A * @L LDA $A2
40 .A *    BNE @L
50 .A *    JMP 'S
60 .D 'S *
RUN
```

This program demonstrates some more advanced assembly features:

- On line 10, you're setting the BASIC variable S, which is to be the start address of the machine language code
- On line 20, 'S tells wAx to insert the value of S as the assembly address after the A command
- On line 30, the * is the "Command Pointer." It is replaced by the *next* address after the last tool that has been executed. So it sort of behaves like the automated prompt in this construction. But *

can be used in any wAx command (see line 60), so it's very useful

- Also on line 30, @L defines a symbol, whose name is @L and whose value is the current address
- On line 40, the @L symbol now appears as an operand, creating a simple loop to the code on line 30
- On line 50, the BASIC variable S makes another appearance, this time as a JMP operand, to go back to the beginning
- Line 60 commands wAx to disassemble the code from the address in variable S to the Command Pointer, right after the JMP instruction

You an go ahead and execute this code with SYS 6400 (or SYS S). Press STOP/RESTORE to exit it.

And now, on line 10, change S = 6400 to S = 6450 and RUN it again. Combining wAx's advanced features allows you to relocate your code. wAx even makes the Command Pointer available to BASIC:

```
PRINT CP
```

You can use wAx in BASIC to automate assembly. This example generates an "unrolled loop":

```
NEW
10 .* 1800
20 .A * LDA #"!"
30 FOR I = 1 TO 10
40 .A * JSR $FFD2
50 NEXT I
60 .A * RTS
RUN
```

```
READY.
.D 1800 *
```

Some new syntax to note:

- In line 10, we're using * as a command to set the Command Pointer, to set the beginning of the code. Can you use **.* 'S**? Absolutely!
- In line 20, we're using a character operand for LDA#. There are several ways to specify immediate operands in wAx

Other Memory Editors

In addition to assembly, wAx has several additional ways to edit memory. Text can be added with quotation marks, and hex values can be entered with : and then up to four hex bytes:

```
.A 1800 LDA #<@T
.A 1802 LDY #>@T
.A 1804 JSR $CB1E
.A 1807 RTS
.A 1808 @T "HELLO, WORLD"
.A 1814 :49 00
.A 1816 {Press RETURN}
```

Also, note that we're using another symbol, this time @T. Here are some interesting things about this symbol usage:

- A symbol always holds a 16-bit address, but you can specify the low or high byte of the value by prefixing the symbol name with < or >

- Notice that the symbol @T is used in operands before it is defined as the address 1808. This is called a "forward reference." When a symbol is used before it's defined, wAx keeps track of where it's used, and then fills those spots in upon the symbol's definition.

wAx also has a binary editor, with one binary byte per line. You can use the Binary Tool, whose command is %, to view binary on one byte per line, with 1s highlighted:

```
.A 0061 %00111100
.A 0062 %01000010
.A 0064 %10100101
.A 0064 %10000001
.A 0065 %10100101
.A 0066 %10011001
.A 0067 %01000010
.A 0068 %00111100
.A 0069 {Press RETURN}
.% 0061 0068
```

There's a lot more, but it'll be covered in the rest of the manual. Hopefully this tutorial has given you some idea how wAx is the finest VIC-20 native assembler that no-money can buy!

6502 Disassembler

To disassemble instructions, use the Disassembler Tool:

```
.D [from] [to]
```

where *from* and *to* are valid 16-bit hexadecimal addresses. If *from* is not provided, wAx will disassemble from the Command Pointer address. If *to* is not provided, wAx will disassemble 16 lines of instructions starting at the specified address. You can display more by pressing RETURN.

If you hold down SHIFT while the disassembly is listing, the disassembly will continue to run until you release SHIFT (or disengage SHIFT LOCK).

The listing will immediately stop if you press the STOP key.

You may cursor up to a disassembled instruction and make changes to it, and press RETURN. wAx will enter the assembler after this.

6502 Assembler

To assemble instructions, enter

```
.A addr mne [operand] [;comment]
```

where *addr* is a valid 16-bit hexadecimal address, *mne* is a 6502 mnemonic, and *operand* is a valid operand. A description of the 6502 instructions is beyond the scope of this document.

wAx will assemble the instruction at the specified address. If the instruction was entered in direct mode, wAx will provide a prompt for the next address after the instruction. You may enter another instruction, or press RETURN.

Note that the comma is an alias for A:

```
., addr mne [operand] [;comment]
```

Comments

wAx stops parsing text after a semicolon, so everything after a semicolon is a comment.

```
.A 1800 AND #$01 ; MASK BIT 0
```

Immediate Mode Operands

wAx supports several types of operands for immediate mode instructions (e.g., LDA #)

```
.A 1800 LDA #$0C        ; HEX BYTE
.A 1802 LDY #"J"        ; PETSCII CHARACTER
.A 1804 LDX #/"A"       ; SCREEN CODE
.A 1806 AND #%11110000  ; BINARY BYTE
.A 1808 CMP #250        ; BASE-10 BYTE
```

Accumulator Mode Operand

wAx supports both explicit and implicit syntax for accumulator mode instructions (ROR, ROL, LSR, ASL):

```
.A 1800 ROR A   ; TAKE YOUR
.A 1802 ROR     ; PICK!
```

Arithmetic

wAx allows you to apply simple arithmetic to address operands by placing + or - after the operand, followed by a hexadecimal digit. The range is -F to +F. For example:

```
.A 1800 STA $9000+5
.A 1803 LDA ($FF-3),Y
.A 1805 STA @V+F
```

Entering Data

For additional details, see Memory Editor.

In addition to 6502 code, you may also enter program data with the @ tool. The following formats are supported:

Text

You may enter quoted text of up to 16 characters:

```
.A 1800 "YOUR TEXT HERE"
```

Hex Bytes

You may enter up to four bytes (in direct mode) or up to eight bytes (in a BASIC program) using a colon:

```
.A 1800 :1A 2B 3C 4D
```

Binary Bytes

You may enter one binary byte (eight bits) using a percent sign:

```
.A 1800 %00110001
```

The Command Pointer

wAx keeps track of the address *after* the last operation for assembly, lists, almost everything. This address, called the Command Pointer, can be inserted into your code with the asterisk (*). This is especially useful when using BASIC programs for assembly. So instead of writing this:

```
10 .A 1800 LDA #"J"
20 .A 1802 JSR $FFD2
30 .A 1805 BRK
```

you can write this:

```
10 .A 1800 LDA #"J"
20 .A *    JSR $FFD2
30 .A *    BRK
```

You may set the address of the Command Pointer by using the * tool:

```
10 *1800
20 .A * LDA #"J"
30 .A * JSR $FFD2
40 .A * BRK
```

An asterisk is replaced with the Command Pointer address in wAx commands, except within quoted strings. If, for some reason, you use it as an operand, remember to use $:

```
.A 1800 JSR $CB1E
.A 1803 JMP $*
```

The CP Variable

When the Command Pointer is changed *during execution of a BASIC program*, wAx sets the CP variable. For example

```
10 .M 1000 1200
20 PRINT "END:",CP-1
```

Note that CP is set to the *next* address that would have been used, so CP-1 will be the *last* address that wAx used.

Symbols

wAx is a “somewhat symbolic assembler,” allowing symbols to represent addresses or values. These symbols can then be used as instruction operands. Addresses and values can be represented as single-character symbols prefixed with the @ sigil. *A symbol name may be a digit (0-9), a letter (A-Z), or an @ sign. There’s also a special symbols, @&, which is always a forward reference (see Forward Reference Symbol, below).*

```
10 .A 1800 LDA #"J"
20 .A *    LDY #22
30 .A * @P JSR $FFD2
40 .A *    DEY
50 .A *    BNE @P
60 .A *    RTS
```

The value of a symbol can be assigned in two ways:

```
.A addr @name
.@name value
```

Where *name* is a valid symbol name, *addr* is a valid 16-bit address, and *value* is a valid 8-bit or 16-bit value.

Symbols can be used before they are assigned to an address (called a "forward reference"), like this:

```
100 .A *    LDA $912D
110 .A *    AND #%01000000
120 .A *    BEQ @2
130 .A *    JMP $EABF
140 .A * @2 LDY $FA
150 .A *    etc...
```

High and Low Bytes

For instructions that require one-byte operands (immediate, zeropage, and X/Y indirect instructions), you may specify the high or low byte of a symbol's value by placing > or <, respectively, immediately before the symbol:

```
10 .@- ; CLEAR SYMBOL TABLE
15 .@ @C 900F ; SCREEN COLOR
20 .@ @V 0314 ; IRQ VECTOR
100 .* 6000
105 .A *     SEI
110 .A *     LDA #<@I   ; LOW BYTE OF IRQ
HANDLER ADDRESS
115 .A *     STA @V
120 .A *     LDA #>@I   ; HIGH BYTE OF IRQ
HANDLER ADDRESS
125 .A *     STA @V+1
130 .A *     CLI
140 .A *     RTS
145 .A * @I
```

```
150 .A *     INC @C
155 .A *     JMP $EABF
```

A symbol may be placed alone on a line, or with an instruction (or data) immediately after it on the same line.

Redefinition

wAx allows symbol redefinition without restriction. If a symbol is defined when it is used as an operand, the operand will be the symbol's value *at the time the symbol is used*. If a symbol is undefined when it is used, it will be recorded as a forward reference, to be filled in when the value is known.

Forward Reference Symbol

While other symbols (@[0-9], @[A-Z], and @@) keep their values after definition, the forward reference symbol (called @&) is cleared whenever it is used as an operand, allowing it to be used as a forward reference multiple times:

```
.A*     BCC @& ; BRANCH TO THE NEXT @&
.A*     BRK
.A* @&
.A*     BEQ @& ; BRANCH TO THE NEXT @&
.A*     BRK
.A* @&
.A*     ORA $FB
.A;     etc...
```

Note that you cannot define @& and use it as an operand in the same line of code. That's why, in this example, definition happens on separate lines.