wAx 2 Assembler

For Commodore VIC-20

Jason Justian
Beige Maze VIC Lab
February 2022

beigemaze.com/wax2

Contents

(continued...)

Appendices

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.

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!

Software License

About wAxpander

wAxpander is a cartridge that includes one 8K ROM chip containing wAx 2 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 the wAxpander cartridge inserted, 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 0  ; RESTART THE VIC-20 AS UNEXPANDED
.U 3  ; RESTART THE VIC-20 WITH 3K EXPANSION
.U 8  ; RESTART THE VIC-20 WITH 8K EXPANSION
.U 16 ; RESTART THE VIC-20 WITH 16K EXPANSION
.U 24 ; RESTART THE VIC-20 WITH 24K EXPANSION
```

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 8K+)
- Regardless of the BASIC expansion configured, the remaining RAM will be available to you

wAx 2 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 the first version of wAx, there are some huge changes in wAx 2. 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 wAx 2, 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 nothing after it will display the next 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 many 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
- Press SHIFT/RETURN

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 {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 can 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 eight 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 {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 is defined, wAx keeps track of where it was 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 {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 Disassemble 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 *(see p. 19)*. 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.

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    ; TAKE
.A 1802 ROR A  ; YOUR
.A 1803 RORA   ; PICK!
```

Arithmetic

wAx allows you to apply simple arithmetic to address operands by placing + or - after the operand, followed by a single 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
.A 1808 LDX #"Z"+1
```

Entering Data

For additional details, see Memory Editor *(p. 33).*

In addition to 6502 code, you may also enter data with the Assemble 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 eight bytes using a colon:

```
.A 1800 :1A 2B 3C 4D 5E 6F 70 81
```

Binary Bytes

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

```
.A 1800 %00110001
```

BASIC Variable Substitution

BASIC variables may be inserted into any wAx command by enclosing the variable name within single quote marks in the command. This feature is subject to the following restrictions:

- Only valid (one- or two-character) floating-point numerical variables are supported (like 'V' or 'VR', but not 'V%' nor 'V$' nor 'VR%')
- Values less than 0 or more than 65535 will result in an ?ILLEGAL QUANTITY error

Values from 0-255 are substituted as one-byte hex values ($00 - $FF). Values from 256-65535 are substituted as two-byte hex values ($0100 - $FFFF).

Here are some examples of BASIC variable substitution:

```
Setting start-of-assembly
10 T = 7000
20 .A 'T' LDA #$00
30 .A *   etc...

Using as character constant
EX = ASC("!")
.A 1800 LDY #'EX'
```

Setting a vector interactively

```
10 INPUT VE
20 .A 1800 STA 'VE'
30 .A *    STY 'VE'+1
```

Viewing a dynamic range of text

```
F = 49344:T = 49536
.I 'F' 'T'
```

Setting a wAx symbol value dynamically

```
A = 48192
.@V 'A'
```

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:

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

You may view the address of the Command Pointer by using the Symbol tool (@), where it is shown next to the asterisk:

```
.@
```

You may explicitly 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.

Note: When you explicitly set the address of the Command Pointer within a BASIC program, the variable UR% (unresolved references) is set to 0 *(see Multi-Pass Assembly, p. 27).*

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 11FF
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 symbol, @&, which is always a forward reference (see Forward Reference Symbol, p. 23).*

```
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 ; These are almost, but not quite
.@name value  ; the same (see p. 25)
```

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, zero page, 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 ; SET COMMAND POINTER (CP)
105 .A *     SEI
110 .A *     LDA #<@I  ; LOW BYTE OF IRQ
115 .A *     STA @V
120 .A *     LDA #>@I  ; HIGH BYTE OF IRQ
125 .A *     STA @V+1
130 .A *     CLI
135 .A *     RTS
140 .A * @I  ; IRQ HANDLER
145 .A *     INC @C
150 .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. 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,* even if it is redefined later. 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
        etc...
```

Note: You cannot define a symbol and use that same symbol as an operand in the same line of code. That's why, in this example, definition happens on separate lines.

Symbol Table Manager

Optionally, wAx maintains a symbol table for use with its somewhat symbolic assembly *(see Symbols, p. 21)*. The @ tool provides a few ways to manage these symbols.

```
.@
```

On a line by itself, @ shows the symbol table. It will show you the following information:

- Defined symbols with the @ sigil
- If followed by a hexadecimal number, the symbol's address
- If followed by a ? and a number, it indicates that the symbol was used as a forward reference and is still undefined. The number indicates the number of times the symbol was used and, thus, how many forward references will be resolved when the symbol's value is defined.
- The last line of the symbol table shows the current Command Pointer address
- If the Command Pointer address is followed by a ? and a number, it indicates the number of forward references that could not be defined because the number of forward references exceeded 12. This means that your code will probably not function properly. If you're entering code in direct mode, you'll be notified of this condition immediately with a ? SYMBOL ERROR. If this condition occurs within a BASIC program, it can be addressed with a second pass *(see Multi-Pass Assembly, p. 27)*.

Setting a Value

There are two ways to set the value of a symbol:

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

where *addr* is a valid 16-bit hexadecimal value, and *name* is a valid symbol name (0~9, A~Z, @, and &), and *value* is a valid 16-bit or 8-bit value or address.

Note: The first form is meant to label code, and will resolve forward references. The second form is designed to set pre-defined constants, and *will not* resolve forward references.

Note: The first form, being meant to label code, sets the Command Pointer to the specified address. The second form, being designed to set pre-defined constants, sets the symbol *without affecting the Command Pointer.* This is so you can set constants within or between sections of code:

```
10 .A 1800 LDA #"J"
20 .@C FFD2
30 .A *     JSR @C
```

Clearing the Symbol Table

```
.@-
```

This clears the symbol table by writing $00 to all symbol table memory. This should be done at the start of a BASIC program that uses symbol-related features.

Symbol Table Limits

See Appendix B: wAx Memory Usage, for more about symbol storage.

Each symbol takes three bytes, and each forward reference record takes three bytes. One additional byte is used to count the number of unresolved forward references since the last Command Pointer set using the * tool. Based on this storage configuration, the symbol table limits are as follows:

- 18 user-defined symbols
- 1 forward reference symbol (@&)
- 12 unresolved forward reference records

Symbol Behavior

The user-defined symbols stay defined until the symbol table is clear. Additionally, the forward reference symbol @& is cleared when it is used as an operand. An unresolved forward reference record is de-allocated (so it can be used again) when its reference is resolved.

Multi-Pass Assembly

Handling Forward Reference Overflow

Direct Mode

It's possible that, during the course of assembly, the forward reference table exceeds the allowed unresolved references. If this happens during direct mode assembly, wAx will throw a ?SYMBOL ERROR. This will alert you that you should examine the symbol table and either

- Resolve one or more forward references
- Clear the symbol table
- Where possible, use the forward reference symbol by changing your operand to @&
- Use a literal address

Multi-Pass Assembly with BASIC

When the forward reference table is exceeded in the course of a BASIC program, wAx does not throw an error. This is because a BASIC program can check for this condition (by testing the variable UR%) and, if necessary, do a second pass, resulting in successful assembly. Such a BASIC program could look like this:

```
10 .@- ; CLEAR SYMBOL TABLE
15 .* 1800
20 FOR I = 1 TO 25
25 .A * LDA @F
30 NEXT I
35 .A * @F BRK
40 IF UR% THEN 15
45 PRINT "DONE!"
```

After 12 iterations of the FOR/NEXT loop, the forward reference table is exceeded, and there's no place to store the 13th-25th references. So wAx increments the BASIC variable UR% (unresolved reference count) for each of those references.

Line 40 checks for unresolved references and, if there are any, goes back for a second pass. Setting the CP at line 15 sets UR% back to 0. On the second pass, the value of the symbol @F is now known, and the assembly may finish.

You won't need to do this check for every program. If you're concerned about the number of forward references, you can check the symbol table after assembly. If the last line of the symbol table has a ? value, like this

```
.@
 * 194e ?02
```

then you have unresolvable references, and you should add the BASIC code to make a second assembler pass.

"Illegal" Instruction Support

To view disassembly with 6502 “illegal” instructions, use the Extended Disassemble tool:

```
.E [from] [to]
```

where *from* and *to* are 16-bit hexadecimal addresses. Other than the display of undocumented instructions, the Extended Disassemble tool behaves exactly like the Disassemble tool.

These instructions may be entered into the assembler alongside the official 6502 instructions, with no special syntax:

```
.A 1800 LAX $EC81 ; LOAD A AND X
.A 1803 BRK
.A 1804 {RETURN}
SYS 6144

BRK
  A--X--Y--P--S--PC--
.;42 42 00 30 F4 1805
```

See Appendix C: Supported "Illegal" Instructions.

Memory Display

Hex Display

To see a hex display listing, use the Hex (or Memory) tool:

```
.M [from] [to]
```

where *from* and *to* are valid 16-bit hexadecimal addresses. If no address is provided, the hex display will continue at the Command Pointer address.

wAx will display memory in hexadecimal bytes, four per line, including a display of the PETSCII characters for a selected range of values. If no *to* is provided, 16 lines of four bytes will be displayed. You can display more by pressing RETURN.

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

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

You may cursor up to a hex display line and change one or more of the values, and press RETURN. wAx will enter the memory editor after this.

Text Display

To see a text display listing with 12 bytes per line, use the Text (or Interpret) tool:

```
.I [from] [to]
```

where *from* and *to* are valid 16-bit hexadecimal addresses. If no address is provided, the hex display will continue at the Command Pointer address.

wAx will display memory as text, 12 characters per line. If no *to* is provided, 16 lines will be displayed. You can display more by pressing RETURN.

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

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

You may cursor up to a text display line and change one or more of the characters, and press RETURN. wAx will enter the memory editor after this.

Binary Display

To see a binary display listing with one byte per line, use the Binary tool:

```
.% [from] [to]
```

where *from* and *to* are valid 16-bit hexadecimal addresses. If no address is provided, the binary display will continue at the Command Pointer address.

wAx will display memory in binary bytes, one byte per line, as 0 and 1. The 1s are displayed in reverse text, so you can easily see character data or binary patterns. If

no *to* is provided, 16 lines will be displayed. You can display more by pressing RETURN.

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

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

You may cursor up to a binary display line and change one or more of the bits, and press RETURN. wAx will enter the memory editor after this.

Memory Editor

wAx offers three kinds of memory editors in addition to the Assembler: A hex editor that can update up to eight values per line, a text editor that can update up to sixteen characters per line, and a binary editor that can update one binary byte per line.

Hex Editor

To edit memory locations as hexadecimal values, use

```
.A addr :nn [nn] [nn] [nn] [nn] [nn] [nn] [nn]
```

where *addr* is a valid 16-bit hexadecimal address, and each *nn* is a valid 8-bit hexadecimal number. You may enter up to eight values on each line.

wAx will update the memory at the specified address. If the values were updated in direct mode, wAx will provide a prompt for the next address after the value list. You may enter additional values, or press RETURN. At each prompt, you may choose any type of editor.

Text Editor

To edit memory locations as a string, use

```
.A addr "string"
```

where *addr* is a valid 16-bit hexadecimal address, and *string* is a PETSCII string up to sixteen characters in length, between double quotes.

wAx will update the memory at the specified address. If the string was updated in direct mode, wAx will provide a prompt for the next address after the end of the string.

You may enter additional values, or press RETURN. At each prompt, you may choose any type of editor.

About Quotation Marks

If you want to use quotation marks within text, you may do so. A quotation mark ends a string only when nothing follows the quotation mark. So, you may do something like this:

```
.A 1800 "I SAID, "HEY!""
```

The closing quotation mark is, in fact, optional. However, you may wish to use a closing quotation mark for aesthetic reasons, or if your text ends in a quotation mark (as above), or to preserve trailing spaces:

```
.A 1800 "HI SCORE:    "
```

Using Screen Codes

wAx will convert PETSCII values to screen code values if you put a slash (/) before the opening quotation mark:

```
.A 1800 /"SCREEN CODES"
```

Note: The M and I commands display PETSCII characters and not screen codes, so you may not see the expected text when using screen codes.

Binary Editor

To edit a memory location as a binary number, use

```
.A addr %bbbbbbbb
```

where *addr* is a valid 16-bit hexadecimal address, and each *b* is a bit value 0 or 1.

wAx will update the memory at the specified address. If memory was updated in direct mode, wAx will provide a prompt for the next address. You may enter additional values, or press RETURN. At each prompt, you may choose any type of editor.

Register Editor

To view the values of registers, processor status, stack pointer, and program counter after the last Go or SYS, use the Register Editor:

```
.R
```

Registers for the next SYS may be set with

```
.;ac [xr] [yr] [pr]
```

where:

ac is the Accumulator as a valid hexadecimal byte

xr is the X Register as a valid hexadecimal byte

yr is the Y Register as a valid hexadecimal byte

pr is the Processor Status Register as a valid hexadecimal byte

Note that you can set the registers for the next SYS or Go by cursoring up to the BRK display:

```
 A__X__Y__P__S__PC__
;12 34 56 78 71 1808
```

The A,X,Y, and P save locations will be set. The stack pointer and program counter will be ignored.

When the register editor is displayed, you're seeing the actual contents of the registers at the time of the last BRK.

However, when you *edit* these values, you're editing the "saved register" locations that BASIC reads and uses to set registers during the execution of the SYS command.

The wAx Go tool (G) leverages BASIC's SYS code, so it will use these values, as well.

Note: Semicolon is an alias for R, so they can be used interchangeably.

Go

To execute code, first set your memory conditions with the Memory Editors, and your register conditions with the Register Editor, then execute the subroutine with the Go tool:

```
.G [addr]
```

where *addr* is a valid 16-bit hexadecimal address. If *addr* is not provided, execution will begin from the current 6502 program counter.

Upon return from the subroutine with RTS, the register display will be shown.

If the code hits BRK instead of RTS, BRK will appear above the register display. In this case, you may continue code execution two bytes after the BRK with

```
.G
```

Note: Use caution when using Go when your program returns with RTS. A proper return address may not be on the stack, and the result is undefined.

Differences Between Go and SYS

Go is designed to behave like you added a breakpoint to the RTS of a subroutine, without having to explicitly set the breakpoint. SYS behaves like SYS.

SYS allows a graceful exit, while Go shows the register display.

Go enables wAx's BRK trapping, while SYS has no impact on the state of the BRK trapping system.

Breakpoint Manager

Setting the Breakpoint

To set the breakpoint, use the Breakpoint Manager tool:

```
.B addr
```

where *addr* is a valid 16-bit hexadecimal address. This will set the breakpoint at the specified address. wAx will update the display to show you the instruction at the specified breakpoint. If the breakpoint set was successful, the instruction will be in reverse text. If the instruction is in regular text, it means you attempted to set a breakpoint in ROM, which cannot be done.

If your breakpoint is shown during disassembly, it will be in reverse text to remind you where the breakpoint is.

When the program encounters a BRK during execution, if the BRK trapping system is enabled, wAx will handle the break by showing the register and program counter display.

Viewing the Breakpoint

To view the breakpoint, use

```
.B
```

If a breakpoint is currently set, the Breakpoint Manager will show the address and code at the breakpoint. If no breakpoint is set, nothing will happen. This operation will also enable wAx's BRK trapping.

Clearing the Breakpoint

To clear the breakpoint, use

```
.B-
```

Entering .B- clears the breakpoint, restores the original breakpoint code, and enables wax's BRK trapping.

Disabling BRK Trapping

When wAx is first started, it sets the BRK interrupt so that it can trap BRK instructions and show the Register Editor.

To disable the BRK trapping system, press STOP/RESTORE. This will turn off breakpoint trapping, but will not clear your breakpoint. If BRK is encountered with the breakpoint trapping system disabled, the BRK will be handled by whatever BRK routine is set in the BRK vector ($0316/$0317), usually the hardware default $FED2.

Note: Successfully updating memory with the Assembler or any Memory Editor will clear the breakpoint.

Note: Using the Memory Save or Load tools will clear the breakpoint.

Disk/Tape/SD Storage

Memory Save

To save a range of memory to disk, tape, or SD card, use the Save tool:

```
.S from to+1 ["filename"]
```

where *from* and *to* are valid 16-bit hexadecimal addresses, and *filename* is a quoted string of up to 12 characters in length. The *filename* may be omitted if you are saving to tape.

The addresses are starting and ending addresses of a memory range. wAx will save the specified memory range to the last-used storage device number. Note that the end address should be the location *after* the last location you want to save.

Note: Memory Save will clear your breakpoint before saving, to restore the code.

Memory Load

To load from disk, tape, or SD card to memory, use the Load tool:

```
.L ["filename"]
```

where *filename* is a quoted string of up to 19 characters in length. The *filename* may be omitted if you are loading from tape. In this case, the next program on tape will be loaded.

The specified program will be loaded into memory using the program's two-byte header. After the load is

complete, the start and end addresses of the program will be displayed.

Note: Memory Load will clear your breakpoint before starting.

File Listing

If the current device is a disk drive or SD card reader, you may obtain a file listing with the File Listing tool:

```
.F
```

You may restrict the display to certain files by providing a partial or complete filename for search:

```
.F "filename or partial filename"
```

The file listing will be formatted with .L, in such a way that you may cursor up to the line and press RETURN to load a file.

Selecting the Device

When wAx is started, it defaults to device #8. You can change the device number by either initiating a BASIC LOAD command followed by pressing STOP, or by setting the device in memory directly:

```
.A 00BA:01
```

Transfer and Fill

Transfer

To copy a block of memory from one location to another, use the Transfer Tool:

```
.T from to target
```

Where *from*, *to*, and *target* are 16-bit hexadecimal addresses. Transfer will copy the specified memory range from the *from* address to the *target* address.

Fill

To fill a block of memory with a specific byte pattern, use the following two-step process. First, set your pattern once at the beginning of the range. This can be any type of pattern, but the example will use the text editor. Second, copy that pattern to the destination using the Command Pointer (*) as the target address. Subtract the pattern length from the end address to make sure the pattern doesn't extend beyond the desired end address.

```
.A from "pattern"
.T from to-minus-length *
```

Search

wAx supports three kinds of memory search: hex search, text search, and code search. The basic syntax for search is:

```
.H addr pattern
```

where *addr* is a valid 16-bit hexadecimal address, and *pattern* is one of three types of search requests.

Search searches for the pattern for 4096 bytes (4K), starting at the specified address. When the search is complete, wAx displays the end address, preceded by .H

To extend the search beyond 4K, hold down the SHIFT key, or engage SHIFT LOCK. The search will continue as long as SHIFT is engaged.

To exit the search before 4K, press STOP. wAx will display the address that would have been searched next if you had not pressed STOP.

Hex Search

```
.H addr:nn [nn] [nn] [nn] [nn] [nn] [nn] [nn]
```

After the address, enter colon, and then up to eight hexadecimal bytes.

Text Search

```
.H addr "string"
```

After the address, enter a quoted string of up to 16 characters in length.

Code Search

```
.H addr mne [operand]
```

After the address, enter a 6502 instruction.

Code Search is a disassembly search that starts disassembling at the specified address, rather than a byte-by-byte pattern search. So the results may differ depending on the address provided.

Compare

To see the addresses of differences and similarities for two regions of memory, use the Compare Tool:

```
.C from-r1 to-r1 from-r2
```

where *from-r1*, *to-r1*, and *from-r2* are valid 16-bit hexadecimal addresses. This will compare the region of memory from *from-r1* to *to-r1* against the region of memory from *from-r2* to (*from-r2* + *to-1* - *from-r1*).

The tool will show a list of addresses in both regions, and the number of consecutive bytes that match (with "=" and green number) or differ (with "!" and red number) starting at those addresses.

Assertion Tester

To test memory-based assertions, use the Assertion Tester tool:

```
.= addr nn [nn] [nn] [nn] [nn] [nn] [nn] [nn]
```

where *addr* is a valid 16-bit address, followed by between 1 and 8 test bytes.

If any of the bytes does not match the byte at the corresponding address, the Assertion Tester tool will return a ?MISMATCH ERROR.

The Assertion Tester tool can be used to generate unit tests in BASIC programs.

Quick Peek

To show two values at an address, use the Assertion Tester tool with only an address:

```
.= addr
```

This will show the value at the address and the following address, in a Memory Editor format.

Numeric Conversion

wAx provides two helpful tools for hex-to-base10 and base10-to-hex conversion.

Hex to Base-10

```
.$ hh[hh]
```

where *hh[hh]* is either an 8-bit or 16-bit hexadecimal value. wAx will display the base10 equivalent of the specified hexadecimal value.

Base-10 to Hex

```
.# base10
```

where *base10* is a positive integer between 0 and 65535. wAx will display the hexadecimal equivalent of the specified value.

BASIC Stage Manager

The Stage Manager lets you allocate memory to one or more BASIC "stages," or memory ranges. This is useful for development in cases where you have a main program under development, and one or more BASIC utility programs to support development.

To change to another BASIC stage, use the Stage Tool:

```
.↑ start-page [end-page] [NEW]
```

where *start-page* and *end-page* are valid 8-bit hexadecimal numbers. They represent the start and end of BASIC memory on page boundaries. If *end-page* is omitted, the end page will be 3.5K after the start page.

```
.↑ 10
```

will reset the stage from $1001 to $1DFF. If you provide *page-end*, the stage will end at the specified page. For example

```
.↑ 60 62 NEW
```

will create a new 512-byte BASIC stage from $6001 to $61FF.

If you add NEW after the page numbers, wAx will set the first three bytes at the stage to $00. If you don't do this before the first use of a stage, the Stage Manager may lock up. If this happens, simply exit with STOP/RESTORE, and try entering the stage again with NEW.

To see the address range (in pages) of the current BASIC stage, enter ↑ on a line by itself:

```
.↑
```

User Plug-Ins

wAx has a vector to a user-defined plug-in program, which can be called with the User Plug-In tool:

```
.U [parameters]
```

parameters is user-defined, based on the plug-in's specification.

The vector to the plug-in is at memory locations $0005 and $0006. When wAx is initialized, this vector is pointed at either the Memory Configuration plug-in (for wAxpander), or the Relocate plug-in (for unexpanded VIC).

A well-constructed wAx 2 plug-in will show a usage template when you enter

```
.U?
```

The template will also appear when you install the plug-in with the Plug-In Manager.

The Plug-In Manager

wAx has a selection of built-in user plug-ins. A menu may be accessed with the Plug-In Manager:

```
.P
```

You may choose which plug-in to install by cursoring up to the plug-in and pressing RETURN. Note that the Plug-In Manager uses only the first two characters to locate the plug-in, so you can install a plug-in with a shortcut at any time. For example,

```
.P "DE" ; Installs Debug plug-in
```

The menu will also show the type of the current plug-in (NORMAL or LIST), the plug-in's address, and the plug-in's usage template.

You may install other plug-ins by providing the plug-in's address:

```
.P [addr]
```

Where *addr* is a 16-bit hexadecimal address. S*ee Developing wAx Plug-Ins, p. 64.*

Included Plug-Ins

The wAx 2 distribution has the following built-in plug-ins available for your convenience:

- *Memory Config* - Restart the VIC-20 in one of five memory expansion configurations *(p. 6)*
- *Relocate* - Update absolute addresses after moving code to a new range *(p. 52)*
- *Debug* - Stop code execution for debugging, and continue running *(p. 53)*
- *ML to BASIC* - Generate BASIC loaders or unit tests for wAx *(p. 56)*
- *Character Helper* - Design custom characters and store them in memory *(p. 58)*
- *BASIC Aid* - Link and renumber BASIC programs *(p. 59)*
- *wAxfer* - High-speed Bluetooth transfer from PC to VIC-20 *(p. 62)*

Plug-In: Relocate

The Relocate plug-in updates addresses within a specified range to values appropriate for another range.

```
Installation
.P "RE"
```

```
Usage:
.U src-from src-to dest-from [dest-to [offset-
override]]
```

where *src-from*, *src-to*, *dest-from*, and *dest-to* are 16-bit hexadecimal addresses. *offset-override* is a 16-bit hexadecimal value.

Starting at *dest-from*, find 3-byte instructions whose operands are between *src-from* and *src-to*, and offset them by the difference between *dest-from* and *src-from*.

Usually, this tool will be used after a transfer operation. For example, to relocate $16 bytes of code from $1800 to $1400, use these two commands:

```
.T 1800 1816 1400
.U 1800 1816 1400
```

You can optionally provide a “destination to” address, which can constrict or expand the range of the code being modified. If you do this, you can also optionally provide an “offset override” value, which will be used in place of the *dest-from* minus *src-from* default.

Plug-In: Debug

The Debug plug-in automatically generates (and, later, removes) a jump to a small, relocated, section of your code. It's basically a breakpoint that can be continued from and traced.

```
Installation
.P "DE"
```

```
Usage
.U [addr]
```

where *addr* is a valid 16-bit hexadecimal address.

When used with an address, Debug copies a few bytes of your code (depending on instruction sizes) to $03EF, and a BRK after the copy. This code snippet, starting at $03EF is called a code "knapsack." Then Debug adds a JMP $03EF at the specified address.

When used without an address, Debug copies the code back from the knapsack to its origin.

Note: A knapsack may not contain a relative branch instruction. If the plug-in encounters a relative branch instruction at or immediately after the breakpoint, it will issue ?UNDEF'D STATEMENT ERROR.

Example

It might be difficult to understand the utility of this, so here's a concrete example:

```
.A 1800 LDY #"A"
.A 1802 TYA
.A 1803 JSR $FFD2
```

```
.A 1806 INY
.A 1807 CPY #"Z"+1
.A 1809 BNE $1802
.A 180B RTS
.A 180C {RETURN}
.G 1800
.ABCDEFGHIJKLMNOPQRSTUVWXYZ
--A--X--Y--P--S--PC--
.;5A 00 5B 33 F8 1800
```

So far so good. We have a program that prints out the alphabet. Now, install Debug and create a knapsack with

```
.P "DE"
.U ADDR
.U 1806 ; At INY
.G 1800
A
BRK
--A--X--Y--P--S--PC--
.;41 00 41 30 F2 03F1
```

At this point, look at the original code with .D 1800 180B. Look at $1806 JMP $03EF. This is Debug's jump to the knapsack code. Now look at the knapsack itself:

```
.D 03EF 03F4
., 03EF BRK       ; The breakpoint
., 03F0 NOP       ; BRK sets PC to $03F1
., 03F1 INY       ; Original code at $1806
., 03F2 CPY #$5B  ; Original code at $1807
., 03F4 JMP $1809 ; Goes back to continue
                  ; original code
```

When this code is jumped to from $1806, it hit the BRK. Program execution pauses, and you may examine registers, code, memory locations, etc. And then, you may resume the program with .G (with no address). Execution starts again from $03F1, and the JMP $1809 brings execution right back to your code. Try this out by entering .G, and watching A and Y increment, and seeing the output from JSR $FFD2 one letter at a time.

When the program finally hits the RTS after displaying Z, the BRK indicator will be absent, and the program will be done.

Once you have the information you need from watching the step-by-step iteration, put your program back together with .U with no address.

Plug-In: ML to BASIC

ML to BASIC converts the 6502 code in a specified range to a BASIC program in the current BASIC stage. The 6502 code will be appended to the existing program, with line numbers in increments of 5.

```
Installation
.P "ML"
```

```
Usage
.U from to+1 [R/H/T]
```

where *from* is the start address, *to* is the end address, and R, H, and T are options.

- If R is specified after the end address, uses the relocatable syntax. Otherwise, uses absolute addresses.
- If H is specified after the end address, creates hex data entry lines instead of 6502 code.
- If T is specified after the end address, creates assertion test data instead of 6502 code.

If the disassembly would extend beyond the end of the BASIC stage, the existing BASIC program (if any) is restored, and an ?OUT OF MEMORY error is displayed.

Example

Enter the following machine language program:

```
.A 1EF0 LDA #<@&
.A 1EF2 LDY #>@&
.A 1EF4 JMP $CB1E
```

```
.A 1EF7 @& "HELLO"
.A 1EFC :00
.A 1EFD {RETURN}
.G 1EF0
HELLO
```

Now we'll use ML to BASIC to bring the code and data into a BASIC program. We'll start by installing the plug-in and adding the code:

```
.P "ML"
.U FROM TO+1 [R/H/T]
NEW
.U 1EF0 1EF7

READY.
.U 1EF7 1EFD H

READY.
LIST
```

Plug-In: Character Helper

The Character Helper plug-in provides a convenient way to design 8x8 custom characters and put them into memory.

```
Installation
.P "CH"
```

```
Usage
.U [addr]
```

where *addr* is a valid 16-bit hexadecimal address.

When used without an address, Character Helper will clear the screen and create an 8x8 grid, with each colon corresponding to a bit in a custom character. Anything except a colon or a space can be used to turn “on” that bit. Cursor around the grid and create your character.

The Character Helper tool may then be used with an address. It will read the grid, convert it into 8 bytes of character data, and store the character at the specified address.

Once you move a character into memory, you may review the data with the Binary Display tool, %.

Plug-In: BASIC Aid

The BASIC Aid plug-in provides two utilities to renumber and link (put together) BASIC programs.

```
Installation
.P "BA"
```

```
Usage
.U R [line [increment]]
.U L stage-page [stage-page]x7
```

Renumber Utility

To renumber the BASIC program in the current stage, use:

```
.U R [line [increment]]
```

where *line* and *increment* are hexadecimal bytes. *line* is the starting line number of the program after the renumber, and *increment* is the value added to the previous line.

If *line* is not provided, the renumbering will start at line 100. If *increment* is not provided, lines will increment by 10.

The Renumber utility can be used to make room for additional lines of assembly if you need to add something.

```
1 .A 1800 LDY #"A"
2 .A * @@ TAY
3 .A *    INY
4 .A *    CPY #"Z"+1
5 .A *    BNE @@
6 .A *    RTS
```

```
.P "BA"
.U R 0A 0A
25 .A *    JSR $FFD2
RUN
SYS 6144
```

Note: This Renumber utility only renumbers lines, as it is designed for easing assembly within BASIC. It will not renumber BASIC operands for GOTO, GOSUB, THEN, etc.

Link Utility

To link one or more BASIC programs from specified BASIC stages to the end of the program in the current BASIC stage, use:

```
.U L start-page [s] [s] [s] [s] [s] [s] [s]
```

where *start-page* and each *s* are 8-bit hexadecimal page numbers indicating the start of a BASIC stage (*see Basic Stage Manager, p. 49*). You may specify up to eight page numbers. The Link utility will add the BASIC program from each specified stage, in order, to the end of the current program.

Note: You may specify each stage any number of times, and you may specify the page of the current stage.

```
.↑ 12 NEW
10 ? "STAGE $12"
15 .A 1800 LDY #8
.↑ 20 NEW
20 ? "STAGE $20"
25 .A * STY $900F
.↑ 2E NEW
10 ? "STAGE $2E"
15 .A * BRK
.↑ 12
```

```
.P "BA"
.U L 20 2E
.U R
LIST
RUN
SYS 6144
```

Notice that here the Renumber utility is run after the three stages are linked into one. This may or may not be warranted, depending on how your lines are numbered.

If you run out of memory in the current stage during the linking process, the Link utility will do these things:

- The entire command will be reverted. This means that if you specify six stages, and run out of memory on the fifth one, Link will revert your program back to its state before the Link utility was invoked.
- The Link utility will display the next memory location in the source stage that *would* have been copied, had memory not run out.
- An ?OUT OF MEMORY error will be displayed.

Note: It's okay to link into an empty stage, to build one program out of other ones.

Note: If you specify a stage without a BASIC program in it, Link will try its best, but you'll probably get junk. Save your nontrivial work!

Plug-In: wAxfer

wAxfer enables high-speed Bluetooth transfer from a PC to your VIC-20, or terminal control of the VIC-20.

This plug-in requires a wAxfer user port device.

```
Installation
.P "WA"
```

```
Usage
.U T    ; Terminal mode
.U B    ; BASIC mode
.U      ; PRG mode
.U addr ; SEQ mode
```

where *addr* is a valid 16-bit hexadecimal address.

Invocation of wAxfer starts its IRQ process and starts listening for data on the user port. When data is received, it can be handled in four ways, depending on the option used

- In Terminal mode wAxfer adds incoming data to the keyboard buffer. Use this if you want to control the VIC-20 from the PC.
- In BASIC mode wAxfer stores the incoming program in the current BASIC stage. Use this if you want to send a BASIC program from the PC.
- In PRG mode wAxfer reads the first two bytes of data, and stores the following data starting at that address. Use this if you want to send a machine language PRG file from the PC.
- In SEQ mode wAxfer will store all of the data it receives, starting at the specified address. Use this if

you want to send a SEQ file (or other raw data file) from the PC.

End Terminal mode with STOP/RESTORE.

End other modes by pressing STOP, or by terminating the transfer at the PC.

For more information about the wAxfer hardware, see https://github.com/Chysn/wAxfer

Developing wAx Plug-Ins

In addition to the wAx user plug-ins included with the system, wAx has a documented API so that programmers can develop their own plug-ins and use them as they would use native wAx tools.

There are two types of plug-in, “Normal” and “List.” A Normal-type plug-in does something and then exits. An example of a Normal-type native tool is the Transfer tool. It performs an operation, and then ends. The Assemble tool is also a Normal-type tool. It performs its task, generates a prompt, and ends. Normal-type tools have quite a bit of diversity in their parameters, even though they *usually* start with a working address.

Examples of List-type native tools include the Disassemble tool and the Memory Display tool. List-type tools always take the same parameters ([from] [to]), and behave in a consistent manner (they start at the Command Pointer when no addresses are provided, they respond to the STOP key, they continue listing with SHIFT engaged, etc.).

When you start designing your plug-in, you’ll want to consider which of the two types is most appropriate for the task you wish your plug-in to perform.

Plug-In Structure

The basic outline of a wAx user plug-in is this:

```
@A JMP @S               ; Jump to code
   : NORMAL | LIST      ; Bit 7 set is List,
                        ; otherwise Normal
   "TEMPLATE"           ; Usually a one-line
                        ; reminder of
```

```
                        ; parameters, but can be
                        ; up to 256 characters
   : 00                 ; The template string is
                        ; $00-terminated
@S ; The plug-in does something useful
   RTS | JMP Next       ; Normal-type plug-ins
                        ; RTS, which will
                        ; generate a wAx prompt;
                        ; List-type plug-ins
                        ; jump to the Next
                        ; endpoint to handle the
                        ; next item
```

The rest of this chapter will deal with "The plug-in does something useful," but first note that plug-ins are installed with the Plug-In Manager tool, using the address of the plug-in:

```
.P addr ; The value of @A above
```

The plug-in is invoked with .U, followed by any parameters it expects. The user may view the template text in two ways:

```
.U?
.P
```

If you use the P command by itself, you'll see the current plug-in's type, address, and the template. U? shows only the template.

A Short Example

```
.@- ; Clear symbol table
.A 1800 JMP @S
        :00             ; Normal-type
        "ADDR"          ; Template
        :00             ; ,,
```

```
@S      BCC @E          ; Carry 0 if no address
        JSR $A01E       ; ResetOut
        JSR $A00F       ; IncAddr
        JSR $A00C       ; HexOut
        JSR $A018       ; PrintBuff
        RTS             ; Return to wAx prompt
@E      JMP $CF08       ; ?Syntax Error
```

Install the plug-in with the Plug-In Manager, then try a few invocations:

```
.P 1800 ; Install plug-in
ADDR    ; Shows the template after installation
.U?     ; Shows the template with help request
ADDR    ; ,,
.U C000 ; Provides a working address
78      ; Gets the value, displays it
.U      ; No address provided, so error out

?SYNTAX
 ERROR
READY.
```

Getting Parameters

When a plug-in is invoked, the wAx API makes an input buffer available. Since nearly all monitor tools require at least a starting 16-bit address, wAx processes the address automatically and places it in the working address pointer. Many plug-ins start by checking for a valid address, which is what the above example does.

The wAx API provides subroutines to get two types of parameters: characters and hexadecimal bytes. Note that wAx does not care about spaces (unless they're

inside quotation marks), so you typically don't need to handle spaces when getting parameters. These invocations are the same, as far as wAx is concerned:

```
.U 1234
.U 12 34
.U 1 2 3 4
```

However, you may wish to differentiate between addresses and hex bytes in documentation, where it makes sense.

You may get the next character with the **CharGet** routine. This will advance the input buffer index and return the next non-space character in the Accumulator. This can be used to set options in your plug-in.

As mentioned above, wAx starts any tool by looking for four hex numerals to create a 16-bit address. This advances the input buffer index past the first four characters of the input buffer. But what if your plug-in doesn't need an address, or needs to use **CharGet** for the first character? That's where **ResetIn** comes in handy:

```
.A 1800 JMP @&
.A 1803 :00
.A 1804 ".U [C]"
.A 180A :00
.A 180B @& JSR $A01B ; ResetIn
.A 180E JSR $A006    ; CharGet
.A 1811 CMP #"C"
.A 1813 BEQ @&
.A 1815 RTS
.A 1816 @& INC $900F
.A 1819 RTS
```

```
.A 181A {RETURN}
.P 1800
.U [C]
.U C
```

Generating Output

The wAx API features a 22-character output buffer that is kept in memory, and printed to the screen all at once. This can be used for providing feedback during the instruction, or for displaying list items. The output routines are:

- *ResetOut* - Starts a new line of output
- *HexOut* - Adds two hex digits (in A), representing one byte, to the output buffer
- *CharOut* - Adds one character (in A) to the output buffer
- *ShowAddr* - Adds the working address to the output buffer
- *ShowCP* - Adds the Command Pointer to the output buffer
- *Disasm* - Adds disassembly at the working address to the output buffer
- *PrintBuff* - Prints the output buffer to the screen
- *PrintStr* - Prints a string (in A,Y), bypassing the output buffer

The general pattern for generating lines of output with the wAx API is:

```
.A * JSR ResetOut  ; Start a new output buffer
.A * LDA #":"      ; To add a character
.A * JSR CharOut   ; ,,
.A * LDA #$4F      ; To add a hex byte
```

```
.A * JSR HexOut    ; ,,
.A * JSR ShowAddr  ; To add the working address
.A * JSR PrintBuff ; Print buffer to the screen
```

This pattern is somewhat different with List-type plug-ins. wAx's list mechanism handles the ResetOut and PrintBuff for you, and they would be omitted for List-type plug-ins.

Memory Usage

For this section, we'll create a List-type plug-in. This will be a disassembler variant. Rather than showing disassembled code, this plug-in will display hex codes, one line per instruction.

When you build your own plug-ins, the VIC-20 is yours and you may use memory however you like. However, wAx earmarks 8 bytes, from $0247 to $024E, for plug-in workspace.

For other memory locations, including the location for the working address and the Command Pointer, see *Appendix B: wAx Memory Usage.*

```
.A 1800 JMP @&
.A 1803 :80         ; List-type plug-in
.A 1804 ".U [FROM] [TO]"
.A 1812 :00
.A 1813 @& LDA #":" ; Show colon
.A 1815 JSR $A009   ;   CharOut
.A 1818 LDY #0      ; Get the current
.A 181A LDA ($A6),Y ;   instruction in A
.A 181C JSR $A033   ; SizeOf inst. A into X
.A 181F STX $0247   ; Save instruction size
.A 1822 @@ LDA #" " ; Show space between bytes
```

```
.A 1824 JSR $A009   ; CharOut
.A 1827 JSR $A00F   ; IncAddr
.A 182A JSR $A00C   ; HexOut
.A 182D DEC $0247   ; Decrement the iterator
.A 1830 BNE @@
.A 1832 JMP $A02D   ; Next
.A 1835 {RETURN}
.P 1800             ; Installation
.U [FROM] [TO]      ; Template reminder
.U 1813 1834
```

List-type plug-ins don't (usually) have variation in their usage templates. So you'll probably always use ".U [FROM] [TO]" as a convention, unless you want to provide additional information. The above plug-in will automatically behave like other List-type tools. That is, it will respond to STOP and SHIFT in the same ways, use and set the Command Pointer in the same ways.

The wAx API

The following subroutines will be useful in the development of user plug-ins.

Note: wAx is located in Block 5 ($A000) for the wAxpander cartridge, but may alternately be located in Block 3 ($6000). For Block 3 versions of wAx, the call addresses start at $6000 instead of $A000.

Addr2CP

Copy working address to Command Pointer

Call Address: **$A027**

Affected Registers: Accumulator

Description: Copies the Working Address, which is usually the current address for a wAx process, to the Command Pointer, which indicates the *next* default address.

If the plug-in is running in a BASIC program, **Addr2CP** will also update the BASIC CP variable.

CharGet

Get character from input

Call Address: **$A006**

Affected Registers: Accumulator, X

Description: Gets the next character from the input buffer. Spaces are not returned unless within quotes. Accumulator of 0 indicates end of input.

CharOut

Add character to output

Call Address: **$A009**

Affected Registers: None

Description: Adds a character with PETSCII code in Accumulator to the output buffer. Note that the output buffer is not printed to the screen until **PrintBuff** is called.

DirectMode

Determine plug-in's execution mode

Call Address: **$A030**

Affected Registers: Y

Description: Sets the Zero flag if the plug-in is currently running in direct mode, and clears the Zero flag if the plug-in is currently running within a BASIC program.

```
JSR DirectMode
BEQ @D ; RUNNING DIRECT
BNE @B ; RUNNING BASIC
```

Disasm

Add disassembly at working address to the output buffer

Call Address: **$A036**

Affected Registers: Accumulator, X, Y

Description: Gets the 6502 instruction at the working address, disassembles it to the output buffer, and advances the working address by the number of bytes for the instruction.

HexGet

Get hexadecimal byte from input

Call Address: **$A003**

Affected Registers: Accumulator, X

Description: If the next two non-space characters in the input buffer represent a valid hexadecimal byte, the value is returned in the Accumulator and the Carry flag is set. Otherwise, the carry flag is clear.

HexOut

Add hexadecimal byte to the output buffer

Call Address: **$A00C**

Affected Registers: Accumulator, X

Description: Adds the value in the accumulator to the output buffer as a two-character hexadecimal number. Note that the output buffer is not printed to the screen until **PrintBuff** is called.

IncAddr

Increment the working address

Call Address: **$A00F**

Affected Registers: Accumulator, X

Description: Sets Accumulator to the value of the working address, then increment the working address. The Accumulator will be the value of the address *prior* to the increment.

IncCP

Increment the Command Pointer

Call Address: **$A012**

Affected Registers: None

Description: Increments the Command Pointer.

Install

Start wAx

Call Address: **$A000**

Affected Registers: Accumulator, X, Y

Description: This is the jump to starting wAx with SYS 40960.

Lookup

Get 6502 instruction data

Call Address: **$A015**

Affected Registers: Accumulator, X, Y

Description: Looks up a 6502 instruction with the opcode in the Accumulator. The Carry flag is set if it's a valid instruction. The instruction's addressing mode is returned in the Accumulator *(see table below)*. The instruction's mnemonic is packed in zero page locations $A4 and $A5, with five bits per letter in bits 0-14.

You may include "illegal" opcodes in the lookup by setting zero page location $AF to value $2C in preparation for the call.

```
Lookup Addressing Mode Table
Mode         Accum.    Example
--------------------------------
INDIRECT     $10       JMP ($0306)
INDIRECT_X   $20       STA ($1E,X)
INDIRECT_Y   $30       CMP ($55),Y
ABSOLUTE     $40       JSR $FFD2
ABSOLUTE_X   $50       STA $1E00,X
ABSOLUTE_Y   $60       LDA $8000,Y
ZEROPAGE     $70       BIT $A2
ZEROPAGE_X   $80       CMP $00,X
ZEROPAGE_Y   $90       LDX $FA,Y
IMMEDIATE    $a0       LDA #$2D
IMPLIED      $b0       INY
RELATIVE     $c0       BCC $181E
ACCUM        $d0       ROR A
```

Next

Handle next list item

Call Address: JMP **$A02D**

Affected Registers: N/A

Description: Next is not a subroutine, it is an endpoint for List-type plug-ins. When the plug-in has finished processing the output for a list item, jump to Next to allow wAx's list system to handle the next list item. Contrast this with Normal-type plug-ins that end with RTS.

PrintBuff

Print output buffer to screen

Call Address: **$A018**

Affected Registers: Accumulator, Y

Description: Print the output buffer to the screen or output device, to a limit of 22 characters. The output buffer is terminated by $00. **PrintBuff** always ends with Reverse Off and Linefeed. Note that PrintBuff does not reset the buffer itself, so a new line should be started with **ResetOut**.

PrintStr

Print a string

Call Address: **$A02A**

Affected Registers: Accumulator, X, Y

Description: Similar to BASIC's $CB1E routine, PrintStr will print a $00-terminated string from address A (low), Y (high). Its limit is 256 characters. The main difference between PrintStr and $CB1E is that PrintStr can be used to print text from Page 2 ($200-$2FF) without corrupting the top of BASIC memory.

ResetIn

Reset input buffer

Call Address: **$A01B**

Affected Registers: Accumulator

Description: Sets the input buffer index back to the beginning. After **ResetIn**, input routines like **CharGet** and **HexGet** will start reading from the beginning of the input buffer.

ResetOut

Reset output buffer

Call Address: **$A01E**

Affected Registers: Accumulator

Description: Sets the output buffer index back to the beginning, to start a new line of output.

ShowAddr

Add working address to the output buffer

Call Address: **$A021**

Affected Registers: Accumulator

Description: Adds the working address to the output buffer as a 16-bit hexadecimal address.

ShowCP

Add Command Pointer address to the output buffer

Call Address: **$A024**

Affected Registers: Accumulator

Description: Adds the Command Pointer address to the output buffer as a 16-bit hexadecimal address.

SizeOf

Return size of 6502 instruction

Call Address: **$A033**

Affected Registers: X

Description: A is passed to SizeOf as the opcode of a legal 6502 instruction. The size of the instruction (1-3) is returned in X. (Caution: For "illegal" instructions, the X will be 1-3, but may not correspond to the actual instruction size.)

Appendix A: wAx Error Messages

wAx adds several additional BASIC error messages, which can occur in both direct mode and during BASIC program execution.

?ASSEMBLY ERROR

You'll get an Assembly Error during any invalid assembly operation during the Assemble command, including bad syntax, unknown instructions, out-of-range operands, etc.

```
Example
.A 1000 STA #$42
```

?MISMATCH ERROR

The Mismatch Error indicates that one or more of the true/false assertions failed during the **=** command. This may indicate that a unit test has failed.

```
Example
.= C000 42
```

?TOO FAR ERROR

The Too Far Error indicates that the relative branch operand is out of range.

```
Example
.A 1800 BCC $4200
```

?CAN'T RESOLVE ERROR

The Can't Resolve Error indicates that a forward definition cannot be resolved. The instruction at the forward definition address is either an illegal, implied, or accumulator mode instruction. Usually this error won't happen unless the code has been changed between the forward definition and the attempted resolution, although it could also occur from corruption of the symbol table.

Example

```
.A 1800 JSR @D
.A 1800 INY    ; NOTE ADDRESS
.A 1801 @D
```

?SYMBOL ERROR

The Symbol Error is a catch-all error for symbolic assembly issues. It can mean

1. The symbol name is invalid. There's a missing or illegal character after the @ sigil. Valid names are letters (A ~ Z), numerals (0 ~ 9), @, and the forward reference symbol @&
2. Too many symbols have been defined. Up to 18 symbols may be defined, and up to 12 unresolved forward definitions can be stored

Example

```
.A 1000 ORA @#
```

Appendix B: wAx Memory Usage

wAx's memory usage is generally unobtrusive and leaves most common storage options open to the programmer, but it is helpful for the programmer to understand wAx's memory footprint. wAx uses the following memory locations:

Command Pointer: $0003-$0004

$0003 and $0004 will be overwritten by most wAx operations to store *, the address of the next instruction or list operation. You can safely overwrite this within machine language programs, as the Command Pointer is usually set explicitly just before implicit use. Mostly, avoid writing to this area during execution of a BASIC program that uses wAx commands.

User Plug-In Vector: $0005-$0006 (optional)

When you use the User Plug-In Tool, wAx executes the code specified by the vector $0005 and $0006. The plug-in can then use wAx's API (or any other user-defined routines) to parse and act on parameters. The user plug-in is initialized when wAx is started. However, if you don't rely on a user plug-in, you can use these locations for your own purposes.

Zero Page Temporary Workspace: $00a3-$00b2

wAx uses this space during the course of most operations. You can use this space in your own programs (usually) without affecting wAx, but such use may affect your ability to inspect these locations accurately *with* wAx.

Temporary Workspace: $0230-$0255

wAx uses this space when a wAx command is issued. Otherwise, this space is used by BASIC as the input buffer. Since wAx commands are always one line or less, wAx can divide this buffer space up for temporary use for its own input and output buffers, and various other things. Once wAx is done with its operation, this area is available for BASIC's normal use.

Plug-In Storage: $0247-$024E

This region of the Temporary Workspace, described above, is set aside for user plug-ins. Feel free to use this memory for your own user plug-ins.

Breakpoint Storage: $0256-$0258 (optional)

wAx stores its breakpoint information here. You can overwrite this if you enter a line of 84 or more characters. So avoid super-long BASIC lines while using the breakpoint features.

Symbol Table: $02A2-$02FF (optional)

wAx stores its symbol table (comprised of symbol names, symbol values, and unresolved forward reference data) here. If you do not use wAx's "somewhat symbolic" features (symbol definition or symbol table initialization), then wAx will leave this range untouched.

Appendix C: Supported "Illegal" Instructions

The following illegal opcodes are supported by wAx. These are recognized by the assembler automatically, and they are disassembled when you use the Extended Disassembler Tool (E).

ANC

- **$0b** ANC #immediate
- **$2b** ANC #immediate

ANE

- **$8b** ANE #immediate

ARR

- **$6b** ARR #immediate

ASR

- **$4b** ASR #immediate

DCP

- **$c7** DCP zero page
- **$d7** DCP zero page,x
- **$cf** DCP absolute
- **$df** DCP absolute,x
- **$db** DCP absolute,y
- **$c3** DCP (indirect,x)
- **$d3** DCP (indirect),y

DOP

- **$04** DOP zero page
- **$14** DOP zero page,x
- **$34** DOP zero page,x
- **$44** DOP zero page
- **$54** DOP zero page,x
- **$64** DOP zero page
- **$74** DOP zero page,x
- **$80** DOP #immediate
- **$82** DOP #immediate
- **$89** DOP #immediate
- **$c2** DOP #immediate
- **$d4** DOP zero page,x
- **$e2** DOP #immediate
- **$f4** DOP zero page,x

HLT (also assembled as JAM)

- **$02** JAM

ISB

- **$e7** ISB zero page
- **$f7** ISB zero page,x

ISB (cont.)

- **$ef** ISB absolute
- **$ff** ISB absolute,x
- **$fb** ISB absolute,y
- **$e3** ISB (indirect,x)
- **$f3** ISB (indirect),y

JAM (also assembled as HLT)

- **$02** JAM

LAE

- **$bb** LAE absolute,y

LAX

- **$a7** LAX zero page
- **$b7** LAX zero page,y
- **$af** LAX absolute
- **$bf** LAX absolute,y
- **$a3** LAX (indirect,x)
- **$b3** LAX (indirect),y

LXA

- **$ab** LXA #immediate

NOP

- **$1a** NOP
- **$3a** NOP
- **$5a** NOP
- **$7a** NOP
- **$da** NOP
- **$fa** NOP

RLA

- **$27** RLA zero page
- **$37** RLA zero page,x
- **$2f** RLA absolute
- **$3f** RLA absolute,x
- **$3b** RLA absolute,y
- **$23** RLA (indirect,x)
- **$33** RLA (indirect),y

RRA

- **$67** RRA zero page
- **$77** RRA zero page,x
- **$6f** RRA absolute
- **$7f** RRA absolute,x
- **$7b** RRA absolute,y
- **$63** RRA (indirect,x)
- **$73** RRA (indirect),y

SAX

- **$87** SAX zero page
- **$97** SAX zero page,y
- **$83** SAX (indirect,x)
- **$8f** SAX absolute

SBC

- **$eb** SBC #immediate

SBX

- **$cb** SBX #immediate

SHA

- **$9f** SHA absolute,y
- **$93** SHA (indirect),y

SHS

- **$9b** SHS absolute,y

SHX

- **$9e** SHX absolute,y

SHY

- **$9c** SHY absolute,x

SLO

- **$07** SLO zero page
- **$17** SLO zero page,x
- **$0f** SLO absolute
- **$1f** SLO absolute,x
- **$1b** SLO absolute,y
- **$03** SLO (indirect,x)
- **$13** SLO (indirect),y

SRE

- **$47** SRE zero page
- **$57** SRE zero page,x
- **$4f** SRE absolute
- **$5f** SRE absolute,x
- **$5b** SRE absolute,y
- **$43** SRE (indirect,x)
- **$53** SRE (indirect),y

TOP

- **$0c** TOP absolute
- **$1c** TOP absolute,x
- **$3c** TOP absolute,x
- **$5c** TOP absolute,x
- **$7c** TOP absolute,x
- **$dc** TOP absolute,x
- **$fc** TOP absolute,x

Appendix D: API Quick Reference

Addr2CP	$A027	*Address to Command Pointer*
CharGet	$A006	*Read Character from Input to A*
CharOut	$A009	*Add Character in A to Output*
DirectMode	$A030	*Set Z If Direct Mode*
Disasm	$A036	*Disassemble at Address to Output*
HexGet	$A003	*Read Hex Byte from Input to A*
HexOut	$A00C	*Add Hex Byte in A to Output*
IncAddr	$A00F	*Address Value to A, Increment Address*
IncCP	$A012	*Increment Command Pointer*
Install	$A000	*Start wAx*
Lookup	$A015	*Lookup 6502 Instruction*
Next	$A02D	*List-type Endpoint*
PrintBuff	$A018	*Print Output Buffer to Screen*
PrintStr	$A02A	*Print String at A,Y*
ResetIn	$A01B	*Reset Input Buffer Index*
ResetOut	$A01E	*Reset Output Buffer Index*
ShowAddr	$A021	*Add Address to Output as Hex*
ShowCP	$A024	*Add CP to Output as Hex*
SizeOf	$A033	*Return Size of 6502 Opcode A in X*

Appendix E: Command Quick Reference

Disassembler

```
.D [from] [to] ; Official 6502 instructions
.E [from] [to] ; Plus "illegal" instructions
```

Assembler/Memory Editor

```
.A addr mne [operand] [;comment]
.A addr "string"
.A addr :nn [nn] [nn] [nn]
.A addr %bbbbbbbb
```

Symbol Manager

```
.* addr      ; Set Command Pointer
.@           ; Show symbol table
.@-          ; Clear symbol table
.@name value ; Set symbol value (8- or 16-bit)
```

Memory Display

```
.M [from] [to] ; Hex
.% [from] [to] ; Binary
.I [from] [to] ; Text
```

Register Editor

```
.R                  ; Display registers
;ac [xr] [yr] [pr]  ; Set registers
```

Go

```
.G addr ; Execute at address
.G      ; Continue after BRK
```

Breakpoint Manager

```
.B addr     ; Set breakpoint, enable trapping
.B          ; Show breakpoint address and inst.
.B-         ; Clear breakpoint, enable trapping
STOP/RESTORE; Disable wAx BRK trapping
```

Disk/Tape/SD Storage

```
.S from to+1 ["filename"]   ; Save
.L ["filename"]             ; Load
.F                          ; Show all files
.F ["partial filename"]     ; Find by name
.A:00BA device              ; Set device number
```

Transfer and Pattern Fill

```
.T from to target    ; Copy
.A from "pattern"    ; Pattern Fill
.T from to *         ; ,,
```

Compare

```
.C from-r1 to-r1 from-r2
```

Search

```
.H addr:nn [nn] [nn] [nn] [nn] [nn] [nn] [nn]
.H addr "string" ; Up to 16 characters
.H addr mne [operand]
```

Assertion Tester

```
.= addr nn [nn] [nn] [nn] [nn] [nn] [nn] [nn]
```

Quick Peek

```
.= addr
```

Numeric Conversion

```
.$ hh[hh]  ; Hex to base-10
.# number  ; Base-10 to Hex
```

BASIC Stage

```
.↑ start-page [end-page] [NEW] ; Set range
.↑                             ; Show range
```

User Plug-Ins

```
.U [varies]     ; Invoke Plug-in
.U?             ; Show usage template
.P              ; Show plug-in Menu and Info
.P "name"       ; Install built-in plug-in
.P addr         ; Install plug-in by address
```