wAx
6502 Assembler
For Commodore VIC-20

Jason Justian
Beige Maze VicLab
June 2020

beigemaze.com/wax

wAx Manual

Table of Contents

About wAx

With your VIC-20 turned off, plug in the wAx cartridge.
Turn on your VIC-20. At the READY prompt, enter ***SYS 24576***

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 most of the usual monitor tools like assembly, disassembly, hex and text data entry, breakpoint management, search, copy and fill, 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," bringing labels and forward references, multi-pass assembly capability, and code relocation. It offers optional built-in support of an extended instruction set in the form of 6502 "illegal" opcodes. It provides for code unit testing with an assertion tester and staging environment selection for multiple BASIC programs in memory. It provides a user tool vector and a documented API for development and integration of custom monitor tools.

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

Be Excellent To Each Other,

Jason Justian
Chysn @ Beige Maze

6502 Disassembler

To disassemble instructions, enter

```
.[addr]
```

where *addr* is a valid 16-bit hexadecimal address. If *addr* is not provided, wAx will disassemble from the persistent counter address.

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.

"Illegal" Opcode Support

To view disassembly with 6502 "illegal" instructions, use

```
,[addr]
```

where *addr* is a 16-bit hexadecimal address. Other than the display of undocumented instructions, the , tool behaves exactly like the . tool.

Appendix C details the undocumented opcodes and mnemonics supported by wAx.

6502 Assembler

To assemble instructions, enter

```
@addr mne [operand] [;comment]
```

where *addr* is a valid 16-bit hexadecimal address, *mne* is a 6502 mnemonic, *operand* is a valid operand, and *comment* is an optional code comment.

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.

Comments

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

```
@1800 AND #$01 ; MASK BIT 0
```

Immediate Mode Operands

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

```
@1800 LDA #$0C        ; HEX BYTE
@1802 LDY #"J"        ; PETSCII CHARACTER
@1804 AND #%11110000  ; BINARY BYTE
@1806 CMP #250        ; BASE-10 BYTE
```

Accumulator Mode Operand

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

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

Entering Data

For additional details, see Memory Editor*, p.16.*

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:

```
@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:

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

Binary Bytes

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

```
@1800 %00110001
```

The Persistent Counter

wAx keeps track of the address *after* the last-assembled instruction. This address 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 @1800 LDA #"J"
20 @1802 JSR $FFD2
30 @1805 BRK
```

you can write this:

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

This allows you to relocate the code by changing its address on the first line.

You may set the address of the persistent counter by using the * tool:

```
 5 *1800
10 @*    LDA #"J"
20 @*    JSR $FFD2
30 @*    BRK
```

An * is replaced with the persistent counter address in wAx commands, except for within the * tool itself, and within quoted strings. If, for some reason, you use it as an operand, remember to use $:

```
@1800 JSR $CB1E
@1803 JMP $*
```

Labels

wAx is a "somewhat symbolic assembler." Addresses can be represented as single-character labels prefixed with the minus sign. *A label name may be a digit (0-9) or letter (A-Z). There is also one special label -@ (see below).*

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

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

```
100 @*    LDA $912D
110 @*    AND #%01000000
120 @*    BEQ -2
130 @*    JMP $EABF
140 @* -2 LDY $FA
150 @;    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 L or H immediately after its label (with no space):

```
 10 @900F -C ; SCREEN AND BORDER COLOR
100 *A000
105 @*     SEI
110 @*     LDA #-IL  ; LOW BYTE OF IRQ HANDLER
115 @*     STA $0314
120 @*     LDA #-IH  ; HIGH BYTE OF IRQ HANDLER
125 @*     STA $0315
130 @*     CLI
```

continued...

```
140 @*     RTS
145 @* -I
150 @*     INC -C
155 @*     JMP $EABF
```

A label 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 its label is used as an operand, the operand will always be the symbol's value *at the time the label is used*. Since developers may lose track of multiple definitions, redefinition is discouraged. If you need a useful convention for local loops, see the special label -@, described in the next section.

Special Label -@

You may use -@ for anything you can use labels for; but it is designed to be a local looping label. -@ can be re-defined and re-used throughout your program as needed, to take the pressure off of symbol table entries:

```
15 @1800 LDY #11
20 @* -@ LDA -D,Y
25 @*    JSR $FFD2
30 @*    DEY
35 @*    BNE -@
40 @*    RTS
50 @* -D "HELLO WORLD"
```

The special label's definition can be cleared during assembly by using the -> label. This allows -@ to be used as a forward reference multiple times:

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

Symbol Table Manager

Optionally, wAx maintains a symbol table for use with its somewhat symbolic assembly (see *Labels*, p. 8). The * tool provides a few ways to manage these symbols.

Setting the Persistent Counter

```
*addr
```

Provide a 16-bit hexadecimal address after * to set wAx's persistent counter to that address. The persistent counter is the next location at which code will be assembled, data will be added, disassembly or memory dump will be displayed, or assertion testing will be performed. All of these tools will also set the persistent counter to the address *after* the last address they use. The * tool lets you set the persistent counter to any address.

All of wAx's tools, except the * tool itself, will replace * with the persistent address, so * can be widely used within tools (although some uses, like .*, are redundant).

*The persistent counter is stored in memory locations $0003 and $0004; if you need to access it within a BASIC program, you can do so as PC=PEEK(3)+PEEK(4)*256*

Displaying the Symbol Table

```
*
```

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

- Defined labels with the label sigil (-)
- If followed by a hexadecimal number, the symbol's address
- If followed by > and a number, it indicates that the label was used as a forward reference and is still undefined. The number indicates the number of times the

label was used and, thus, how many forward references will be resolved when the symbol's address is defined.

- The last line of the symbol table shows the current persistent counter address
- If the persistent counter 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. 12).

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 several labels.

The wAx symbol table is stored between $02a2 and $02ff. This memory is not addressed at all by wAx unless you use the somewhat symbolic features. The persistent counter is stored at locations $0003 and $0004, and can be used without affecting the symbol table area.

Multi-Pass Assembly

Symbol Table Limits

wAx stores the symbol table between $02a2 and $02ff. Each label 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 persistent counter set using the * tool. Based on this storage configuration, the symbol table limits are as follows:

- 18 user-defined labels
- 1 special label (-@)
- 12 unresolved forward reference records

Symbol Behavior

- The user-defined labels stay defined until the symbol table is clear.
- The special label can be cleared (for use as a forward reference) with -> as an operand.
- Unresolved forward reference records are de-allocated when the the reference is resolved.

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
- Use the special label by changing your operand to ->
- Use a literal address

Multi-Pass Assembly with BASIC

When the forward reference table is exceeded in a BASIC program, wAx does not throw an error. The program can check for this condition and do a second pass, resulting in successful assembly.

```
100 *-            ; CLEAR SYMBOL TABLE
105 *1800
110 @* LDA #-0L   ; FIRST FORWARD REFERENCE
115 @* LDA #-0H   ; SECOND
120 @* LDX -1     ; THIRD...
etc...
199 @* JMP -C     ; THIRTEENTH
200 @* -0 "RESOLVE 1ST&2ND"
201 @* -1         ; RESOLVE THIRD
etc...
299 @* -C         ; RESOLVE THIRTEENTH
300 IF PEEK(767) THEN 105
305 PRINT "DONE!"
```

At line 199, the forward reference table is exceeded, and there's no place to store the 13th reference. So wAx increments memory location $02ff (767). The program resolves the forward references from line 200 on. But the reference at 299 only sets the address of -C without resolving any forward references, because wAx is not aware of that reference.

That's where line 300 comes in. If location 767 is set, then we should go back for another pass. Line 105 resets the persistent counter to $1800, and sets location 767 back to 0. On this pass, when BASIC gets to line 199, the address of -C is known from the first pass, and the operand for the JMP will be set properly. When we get to line 300 again, the value of location 767 should now be 0, and the assembly can finish.

You won't need to do this check for every program. 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.

Undocumented Instruction Support

To view disassembly with 6502 "illegal" instructions, use

```
,[addr]
```

where *addr* is a 16-bit hexadecimal address. Other than the display of undocumented instructions, the , tool behaves exactly like the . tool (see *6502 Disassembler*, p. 4).

To assemble with these instructions, just enter them into the assembler with the @ tool:

```
@1800 LDY #$42
@1802 STY $FA
@1804 LAX $FA
@1806 BRK
SYS6144

 A__X__Y__P__S__PC__
;42 42 42 30 F8 1808
```

Appendix C contains a list of "illegal" opcodes and mnemonics supported by wAx.

Memory Dump

Hex Dump

To see a hex dump listing, enter

```
:[addr]
```

where *addr* is a valid 16-bit hexadecimal address. If no address is provided, the memory dump will continue at the persistent counter address.

wAx will display memory in hexadecimal bytes, four per line, including a display of the PETSCII characters for a selected range of values. 16 lines of four bytes will be displayed. You can display more by pressing RETURN.

If you hold down SHIFT while the memory dump is listing, the memory dump 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 dump line and change one or more of the values, and press RETURN. wAx will enter the memory editor after this.

Binary Dump

To see a binary dump listing with one byte per line, enter

```
%[addr]
```

where *addr* is a valid 16-bit hexadecimal address. If no address is provided, the binary dump will continue at the persistent counter address.

Control of the output is the same as that of the Hex Dump tool.

Memory Editor

wAx offers three kinds of memory editors: A hex editor that can update up to four 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.

Hex Editor

To edit memory locations as hexadecimal values, use

```
@addr :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 four values on each line in direct mode, and up to eight values in a BASIC program.

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 a different editor.

Text Editor

To edit memory locations as a string, use

```
@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 a different editor.

Binary Editor

To edit a memory location as a binary number, use

```
@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 the value was updated in direct mode, wAx will provide a prompt for the next address. At each prompt, you may choose a different editor.

Register Editor

To view the register values from last SYS or subroutine execution, either by BRK or RTS, use

```
;
```

Registers for the next SYS or subroutine execution 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 Subroutine Execution (←) by cursoring up to the BRK display:

```
 A__X__Y__P__S__PC__
;12 34 56 31 F2 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.

When you *edit* these values, you're editing the "register storage" locations that BASIC reads and uses to set registers during the execution of the SYS command. The wAx ← tool (Subroutine Execution) use these values, as well.

Subroutine Execution

To execute a subroutine, first set your memory conditions with the Memory Editor, and your register conditions with the Register Editor, then execute the subroutine with

```
←addr
```

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

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

```
←
```

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

To view and/or edit registers before executing a subroutine, enter the left arrow on a line by itself.

Differences Between ← and SYS

- ← 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 ← shows the register display upon return
- ← enables wAx's BRK trapping, while SYS has no impact on the state of the BRK trapping system
- If you're generating unit tests with wAx in BASIC, you'll want to use SYS. If you want to immediately see the register and memory results of a subroutine, you may want to use ←
- SYS takes a decimal literal or variable address, while ← takes a hexadecimal address literal

Breakpoint Management

Set a Breakpoint

To set the breakpoint, use

```
![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.

Clearing the Breakpoint

To clear the breakpoint, use

```
!
```

Entering ! on a line by itself clears the breakpoint (and restores the original breakpoint byte), but enables BRK trapping.

Successful update of memory with the assembler or memory editor, or use of the Save or Load Memory tools, will also clear the breakpoint.

Disable Breakpoint Trapping

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 off, the BRK will be handled by whatever BRK routine is set in the BRK vector ($0316/$0317), usually the hardware default $FED2.

Memory Save and Load

Save

To save a range of memory to disk, tape, or SD, use

```
>start end+1 ["filename"]
```

where *start* and *end* 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.

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.

Load

To load from disk, tape, or SD to memory, use:

```
<["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. If you loaded from disk, the start and end addresses of the program will be displayed.

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 address in memory:

```
@00BA:01
```

Copy and Fill

To copy a block of memory from one location to another, use

```
&start end target
```

Where *start*, *end*, and *target* are 16-bit hexadecimal addresses.

Pattern 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 persistent counter (*):

```
@start "pattern"
&start end-pattern_length *
```

Search

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

```
/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 /

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

```
/addr:nn [nn] [nn] [nn] [nn] [nn] [nn] [nn]
```

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

Text Search

```
/addr "string"
```

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

Code Search

```
/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.

Assertion Tester

To test memory-based assertions, use

```
=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 = command will return a ?MISMATCH ERROR.

The Assertion Tester can be used to generate unit tests of your subroutines within BASIC programs:

```
10 REM TEST AN ML SORT ROUTINE
15 REM Y=START HIGH, X=START LOW, A=LENGTH
20 ;18 00 08
25 REM UNORDERED DATA
30 @1800:44 3F 0B 92 FD 45 AC 01
40 REM RUN ASCENDING SORT
45 SYS 780
50 REM TEST RESULTS
55 =1800 01 0B 3F 44 45 92 AC FD
60 PRINT "SUCCESS!"
```

If your sort routine correctly sorts the data, the program will finish and announce success. If there's a problem with the sort routine, and it incorrectly sorts the data, the test will fail with ?MISMATCH ERROR.

Hex/Base-10 Converters

wAx provides two helpful tools for hex-to-base-10 and base-10-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 base-10 equivalent of the specified hexadecimal value.

Base-10 to Hex

```
#base-10
```

where *base-10* is a positive integer between 0 and 65535. wAx will display the hexadecimal equivalent of the specified value.

BASIC Stage Select

wAx provides a tool that 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

```
↑ 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 (or if it is lower than the start page), then the end page will be 3.5K after the start page. For example

```
↑ 10
```

will reset the stage back to the default of an unexpanded VIC-20. If you provide *page-end*, the stage will end at the specified page. For example

```
↑ A0 A2 NEW
```

will create a new 512-byte BASIC stage at $a000.

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 tool may lock up. If this happens, simply intervene with STOP/RESTORE, and try entering the stage again with NEW.

If you enter ↑ on a line by itself, wAx will display the start address of the current BASIC stage.

User Tool

wAx has a vector to a user-defined tool, which can be called with

```
'[parameters]
```

parameters is user-defined, based on the user tool's requirements.The vector to the user tool is at memory locations $0005 and $0006. When wAx is initialized, this vector is pointed at the wAx installation routine (so entering ' will restart wAx).

Developing User Tools

You can use memory locations and subroutines in wAx's API to develop user tools. When a wAx tool is invoked, wAx checks for a valid 16-bit Effective Address (EA) immediately after the tool's character. If a valid EA is found the Carry flag is set, and the EA is stored in $a6 (low byte) and $a7 (high byte). If the Carry flag is clear, either the EA was invalid or absent.

Spaces are ignored during the parsing of hex parameters, so you may treat the EA (and additional parameter bytes) as addresses or bytes, depending on your tool's requirements.

Some tools use the Persistent Counter (PC) in the absence of an EA. The PC is stored in $03 (low byte) and $04 (high byte). If a tool uses the PC, the PC should be transferred to the EA, and then the PC should be updated when the tool has completed its work.

The wAx API

The following subroutines may be useful in the development of user tools.

Routine: Address
Call Address: $6B72
Affected Registers: Accumulator
Description: Adds the Effective Address to the output buffer as a 16-bit hexadecimal number.

Routine: Buff2Byte
Call Address: $6ABF
Affected Registers: Accumulator, Carry flag, 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.

Routine: CharGet
Call Address: $6AB5
Affected Registers: Accumulator, X
Description: Get the next character from the input buffer. Spaces are not returned unless within quotes. Accumulator of 0 indicates end of input.

Routine: CharOut
Call Address: $6B1E
Affected Registers: Accumulator
Description: Adds the character in the Accumulator to the output buffer.

Routine: CurrValue
Call Address: $6B05
Affected Registers: Accumulator, X
Description: Returns the value at the Effective Address in the Accumulator. X is set to 0.

Routine: Hex
Call Address: $6B32
Affected Registers: Accumulator
Desciption: Adds the number in the Accumulator to the output buffer as an 8-bit hexadecimal number.

Routine: NextValue
Call Address: $6AFF
Affected Registers: Accumulator, X
Description: The Effective Address is incremented by 1, and the Accumulator is set with the new EA's value. X is set to 0.

Routine: PrintBuff
Call Address: $6C70
Affected Registers: Accumulator, Y
Description: Flush the output buffer to the screen or output device. The output buffer's length should not exceed 22 characters.

Routine: ResetIn
Call Address: $6C41
Affected Registers: Accumulator
Description: Resets the input buffer

Routine: ResetOut
Call Address: $6C3C
Affected Registers: Accumulator
Description: Resets the output buffer

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 @ command, including bad syntax, unknown instructions, out-of-range operands, etc.

```
Example
@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
@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 or immediate mode instruction. Usually this error won't happen unless the code has been changed between the forward definition and the attempted resolution.

```
Example
@1800 JSR -D
@1800 INY
@1801 -D
```

?SYMBOL ERROR

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

- The label is invalid. There's a missing or illegal character after the label sigil (minus sign). Valid labels are digits (0-9), letters (A-Z), the special label -@, and the special forward reference redefinition ->
- Too many symbols have been defined. Up to 18 labels may be defined, and up to 12 unresolved forward definitions can be stored

```
Example
@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:

Persistent Counter: $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 persistent counter is usually set explicitly just before implicit use. Mostly, avoid writing to this area during a BASIC program doing assembly.

User Tool Vector: $0005 - $0006

The address of the user tool (').

Zeropage Workspace: $00a3-$00ae

wAx uses this space during the course of operations. You can use this space in your own programs (usually) without affecting wAx, but wAx may overwrite this range when you use it.

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, Once wAx is done with its operation, this area is available for BASIC's normal use, or your use.

Breakpoint Storage: $0256-$0258

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 label names, label addresses, and unresolved forward reference data) here. If you do not use wAx's "somewhat symbolic" features (labels in your code, symbol table initialization with *-), then wAx will leave this range untouched.

Appendix C: 6502 "Illegal" Instruction Support

The following illegal opcodes are supported by wAx. These are recognized by the assembler automatically, and they are disassembled when you use the , tool.

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

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

ARR
- $6b ARR #immediate

ASR
- $4b ASR #immediate

LXA
- $ab LXA #immediate

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

SBX
- $cb SBX #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

ISB
- $e7 ISB zero page
- $f7 ISB zero page,x
- $ef ISB absolute
- $ff ISB absolute,x
- $fb ISB absolute,y
- $e3 ISB (indirect,x)
- $f3 ISB (indirect),y

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

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

SBC
$eb SBC #immediate

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

SHX
$9e SHX absolute,y

SHY
$9c SHY absolute,x

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

ANE
$8b ANE #immediate

SHS
$9b SHS absolute,y

JAM (also HLT)
$02 JAM

Appendex D: wAx License (Including this Document)

Software:

Manual and Cover Photograph:

Appendix E: Quick Reference

Disassemble from Address

```
.addr             ; Official 6502 instructions
,addr             ; Plus Undocumented instructions
```

Assemble at Address

```
@addr mne [operand]
@addr "string"
@addr:nn [nn] [nn] [nn]
@addr%bbbbbbbb
```

Symbol Management

```
*addr             ; Set persistent counter
*                 ; Show symbol table
*-                ; Clear symbol table
```

Memory Dump

```
:addr ; Hex
%addr ; Binary
```

Register Management

```
←                 ; Display registers
```

Execute Subroutine

```
←addr
```

Breakpoint Management

```
!addr             ; Set breakpoint, enable BRK trapping
!                 ; Clear breakpoint, enable BRK trapping
STOP/RESTORE      ; Disable BRK trapping
```

Memory Save/Load

```
>start end+1 ["filename"]     ; Save
<["filename"]                 ; Load
```

Copy and Pattern Fill

```
&start end target             ; Copy
@start "pattern"              ; Pattern Fill
&start end-length *           ; ,,
```

Search

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

Assertion Tester

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

Numeric Conversion

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

BASIC Stage Select

```
↑start-page [end-page] [NEW]  ; Set stage address
↑                             ; Show stage address
```

User Tool

```
`[parameters]
```